บทที่ 5
อรรถศาสตร์เบื้องต้น (Basic Semantics)

5.1 ลักษณะประจำ การผูกโยง และฟังก์ชันความหมาย
(Attributes, Binding, and Semantic Functions)

5.2 การประกาศ บล็อก และสโคป
(Declarations, Blocks, and Scope)

5.3 ตารางสัญลักษณ์
(The Symbol Table)

5.4 การจัดสรร การมีอายุ และสิ่งแวดล้อม
(Allocation, Extent, and the Environment)

5.5 ตัวแปร และตัวคงที่
(Variables and Constants)

5.6 สมนาม การนำมาใช้ไม่ได้ และขยะ
(Aliases, Dangling References, and Garbage)

5.7 การประเมินผลนิพจน์
(Expression Evaluation)

แบบฝึกหัด

บทที่ 5
อรรถศาสตร์เบื้องต้น (Basic Semantics)

ในบทนี้ จะได้แนะนำคุณสมบัติที่สำคัญของอรรถศาสตร์ของภาษาโปรแกรม

การอธิบาย (specifying)) อรรถศาสตร์ ของภาษาโปรแกรมเป็นงานซึ่งยากกว่า การอธิบายวากยสัมพันธ์ เช่น เราอาจคาดการณ์ได้เมื่อเราพูดเกี่ยวกับ ความหมาย (meaning) ซึ่งตรงกันข้ามกับรูปแบบ หรือโครงสร้าง (form or structure)

การอธิบายอรรถศาสตร์ มีหลายวิธี เช่น

1) **โดยคู่มือของภาษา** (By a language reference manual) เป็นวิธีซึ่งใช้กันมากที่สุด ประสบการณ์ของการอธิบายด้วยภาษาอังกฤษ ทำให้ คู่มือของภาษา ชัดเจนมากกว่า ถูกต้องมากกว่า เป็นเวลาผ่านมาหลายปี แต่วิธีนี้ยังคงยากลำบาก จากการขาดความถูกต้อง ซึ่งยึดติด ในการอธิบายด้วยภาษาธรรมชาติ และยังอาจมีความไม่ครบถ้วน และความกำกวมได้ (and also may have omissions and ambiguities)

2) **โดยการนิยามตัวแปลภาษา** (By a defining translator)

วิธีนี้มีข้อดีที่ว่า คำถามต่างๆ เกี่ยวกับภาษา สามารถตอบได้โดยการทดลอง (เช่น ในวิชาเคมี หรือ ฟิสิกต์) แต่มีอุปสรรคคือ คำถามเกี่ยวกับ พฤติกรรมของโปรแกรม (program behavior) ไม่สามารถตอบได้ ในวิชาการขั้นสูง นั่นคือ เราต้องกระทำการ (execute) โปรแกรม เพื่อค้นหาว่า มันทำอะไร อุปสรรคอีกอย่างหนึ่งคือ **ที่ผิด** (bugs) และการยึดติดอยู่กับเครื่องคอมพิวเตอร์ (machine dependencies) ในตัวแปลภาษา ซึ่งกลายเป็นส่วนต่างๆ ของ อรรถศาสตร์ของภาษา อาจจะเป็นไปโดย ไม่ตั้งใจ นอกจากนี้แล้ว ตัวแปลภาษา อาจจะเคลื่อนย้ายไม่ได้ บนคอมพิวเตอร์ทุกเครื่อง และอาจจะใช้ไม่ได้ทั่วไป

3) **โดยบทนิยามแบบทางการ** (By a formal definition)

เช่น วิธีคณิตศาสตร์ ซึ่งถูกต้อง แต่วิธีนี้ซับซ้อนและเป็นนามธรรม (complex and abstract) และต้องมีการศึกษาเพื่อทำความเข้าใจ วิธีแบบทางการต่างๆ กันมีให้ใช้อยู่ การเลือกวิธีใดนั้นขึ้นอยู่กับว่า ตั้งใจจะใช้ทำอะไร บางทีวิธีทางการแบบที่ดีที่สุด ซึ่งใช้สำหรับอธิบาย การแปล และ การกระทำการของโปรแกรม คือ **อรรถศาสตร์เชิงรายละเอียด** ซึ่งอธิบายอรรถศาสตร์โดยใช้ชุดของฟังก์ชัน

(Perhaps the best formal method to use for the description of the translation and execution of programs is **denotational semantics,** which describes semantics using a series of functions.)

ในบทนี้ เราจะใช้ การปรับ (adaptation) ของ การอธิบายแบบไม่เป็นทางการ เช่นที่มีอยู่ในคู่มือ รวมกับการใช้ฟังก์ชัน อย่างง่ายเหมือนในการอธิบายเชิงรายละเอียด โดยจะให้การนิยามนามธรรมของการดำเนินการ ซึ่งเกิดขึ้น ในระหว่าง การแปล (translation)) และการกระทำการ (execution) ของ โปรแกรมโดยทั่วๆ ไป ไม่ว่าจะเป็นภาษาอะไรก็ตาม แต่จะเน้นบนรายละเอียดของ ภาษาเหมือนอัลกอล (Algol-like languages) เช่น ภาษา C, Pascal, Modula-2, และ Ada

5.1 ลักษณะประจำ การผูกโยง และ ฟังก์ชันความหมาย

(Attributes, Binding, and Semantic functions)

กลไกของการนิยามนามธรรมขั้นพื้นฐาน ในภาษาโปรแกรม หมายถึง การใช้ ชื่อ หรือ ไอเดนติไฟเออร์ เพื่อแทน เอนทิตี หรือ ตัวสร้าง ของภาษา

(A fundamental abstraction mechanism in a programming language IS the use of **names,** or **identifiers.** to denote language entities or constructs.)

ในภาษาส่วนใหญ่ ตัวแปร โปรซีเดอร์ และตัวคงที่ สามารถมี ชื่อ ซึ่ง โปรแกรมเมอร์เป็นผู้กำหนดให้ ขั้นตอนหลักในการอธิบาย อรรถศาสตร์ ของภาษา คือ การอธิบาย ข้อตกลง ซึ่งบอกความหมาย ของ ชื่อแต่ละชื่อที่ใช้ในโปรแกรม

(A fundamental step in describing the semantics of a language IS to describe the conventions that determine the meaning of each name used in a program)

นอกเหนือจาก ชื่อ แล้ว การอธิบาย อรรถศาสตร์ ของภาษาโปรแกรม จำเป็น ต้องเข้าใจแนวคิดของ ตำแหน่ง (location) และ ค่า (value)

ค่า หมายถึง ปริมาณใดๆ ซึ่งเก็บไว้ได้ เช่น จำนวนเต็ม จำนวนจริง หรือ ค่าของแถวลำดับ ซึ่งประกอบด้วย ลำดับของค่าต่างๆ ซึ่งเก็บ ณ ดรรชนี แต่ละตัวของแถวลำดับ

(Values are any storable quantities, such as the integers, the reals, or even array values consisting of a sequence of the values stored at each index of the array.)

ตำแหน่ง คือ สถานที่ซึ่งเก็บค่า ตำแหน่งเหมือนกับเลขที่อยู่ ในหน่วยความจำของคอมพิวเตอร์ แต่เราสามารถคิดว่า มันเป็นนามธรรมมากกว่า เลขที่อยู่ของคอมพิวเตอร์ เครื่องใดเครื่องหนึ่ง

(Locations are places where values can be stored. Locations are like address in the memory of a computer, but we can think of them more abstractly than as the address of a particular computer.)

ถ้าจำเป็น เราสามารถคิดว่า ตำแหน่ง คือเลข จำนวนเต็ม ตั้งแต่ 0 ไปจนถึง เลขตำแหน่งสูงสุด

ความหมายของชื่อ บอกได้โดยคุณสมบัติ หรือ **ลักษณะประจำ** ที่เกี่ยวข้องกับ ชื่อ นั้น

(The meaning of a name is determined **by the** properties. or **attributes** associate to the name.)

ตัวอย่าง การประกาศของ Pascal

```
const  n = 5;
```

หมายถึง ทำให้ n เป็นตัวคงที่ มีค่าเท่ากับ 5 นั่นคือ เกี่ยวข้องกับชื่อ n ด้วยลักษณะประจำสองสิ่งคือ const กับ "value 5"

ตัวอย่าง การประกาศของ Pascal

```
var   x : integer;
```

หมายถึง การเกี่ยวข้องของ ลักษณะประจำ var และข้อมูลชนิด integer กับชื่อ x

ตัวอย่าง การประกาศของ Pascal

```
function f(n : integer) : boolean;
begin


end;
```

หมายถึง การเกี่ยวข้อง ของลักษณะประจำ function ให้กับ ชื่อ f และ ลักษณะประจำ

อื่นๆ ต่อไปนี้

1. จำนวน ชื่อ และแบบชนิดข้อมูล ของ พารามิเตอร์ของมัน (ในกรณีนี้ มีพารามิเตอร์ หนึ่งตัว ชื่อ n และเป็นข้อมูลชนิด **integer**)

   (The number, names. and data types of its parameters (in this case, one parameter with name n and **data** type **integer))**

2. ชนิดข้อมูลของค่าส่งกลับ (ในกรณีนี้ เป็นข้อมูลชนิดบูลีน)

   (The data type of its returned value (in this case, **boolean**))

3. body ของรหัส ซึ่งจะถูกกระทำการ เมื่อ f ถูกเรียก (ในกรณีนี้ เราไม่ได้เขียนรหัส เพียงแต่แสดงด้วยจุดสามจุด)

   (The body of code to be executed when f is called (in this case, we have not written this code but Just indicated it with three dots))

การประกาศไม่ได้เป็นเพียง ตัวสร้างภาษา (language constructs) ซึ่งเกี่ยวข้องกับลักษณะประจำที่ให้กับ ชื่อ เท่านั้น

(Declarations are not the only language constructs that can associate attributes to names.)

ตัวอย่าง การกำหนดค่า

```
x := 2;
```

หมายถึง ตัวแปร x เกี่ยวข้องกับ ลักษณะประจำตัวใหม่คือ "value 2"

ตัวอย่าง ถ้า y เป็นตัวแปรชนิด pointer ประกาศ ดังนี้

```
var y : ^integer;
```

ข้อความสั่ง

```
new(y);
```

หมายถึง การจัดสรรหน่วยความจำ ให้กับตัวแปรชนิดจำนวนเต็ม นั่นคือ การเกี่ยวข้องของลักษณะประจำ ซึ่งเป็นตำแหน่งให้กับ ตัวแปรจำนวนเต็ม และตำแหน่งนี้คือ y^ นั่นคือเกี่ยว

กับลักษณะประจำตัวใหม่ ให้กับ y

(allocates memory for au integer variable, that is, associates a location **attribute** to it and assigns this location to **y^**, that is, associates a new value attribute to **y**.)

การผูกโยง และเวลาของการผูกโยง

(Binding and Binding Time)

กรรมวิธีของการเกี่ยวข้องกัน ของลักษณะประจำ หนึ่งอย่าง กับ ชื่อ เรียกว่า **การผูกโยง**

(The process of associating an attribute to a name is called **binding**.)

ในบางภาษา ตัวสร้าง (constructs) ซึ่งทำให้ค่าต่างๆ ผูกโยงกับ ชื่อ (เช่นในตัวอย่างการประกาศ ของ Pascal ข้างต้น) ความจริง เรียกว่า การผูกโยง ไม่ใช่ การประกาศ

การแบ่งประเภท ของ ลักษณะประจำ ขึ้นอยู่กับเวลาระหว่างกระบวนการแปลโปรแกรม หรือกระบวนการกระทำการโปรแกรม เมื่อมีการคำนวณ และมีการผูกโยง ให้กับชื่อ สิ่งนี้ เรียกว่า **เวลาการผูกโยง** (binding time) ของลักษณะประจำ

เวลาผูกโยง โดยทั่วไปแบ่งออกเป็นสองชนิด คือ

การผูกโยงแบบคงที่ (static binding)

การผูกโยงแบบพลวัต (dynamic binding)

การผูกโยงแบบคงที่ เกิดขึ้นก่อนการกระทำการ ในขณะที่ การผูกโยงแบบพลวัต เกิดขึ้นระหว่าง การกระทำการ ลักษณะประจำ ซึ่งผูกโยงอย่างคงที่ คือ **ลักษณะประจำแบบคงที่** (static attribute) ส่วนลักษณะประจำ ซึ่งผูกโยงอย่างพลวัต เรียกว่า **ลักษณะประจำแบบพลวัต** (dynamic attribute)

ภาษาซึ่งลักษณะประจำ ผูกโยงอย่างคงที่ จะแตกต่างอย่างมากจากภาษา ซึ่ง ลักษณะประจำผูกโยงอย่างพลวัต บ่อยครั้งที่ **ภาษาเชิงหน้าที่** (functional languages) จะมีการผูกโยงแบบพลวัต มากกว่า **ภาษาเชิงคำสั่ง** (imperative languages)

เวลาของการผูกโยง ยังขึ้นอยู่กับตัวแปลภาษาด้วยเช่นกัน ตัวแปลคำสั่งโดยบทนิยาม กระทำการผูกโยงทั้งหมด อย่างพลวัต ในขณะที่ ตัวแปลโปรแกรม จะกระทำ การผูกโยงส่วนใหญ่อย่างคงที่

(Binding times can also depend on the translator. Interpreters by definition perform all bindings dynamically, while compilers will perform many bindings statically.)

เพื่อทำให้ การอภิปราย หัวข้อ ลักษณะประจำ และการผูกโยง เป็นอิสระจาก หัวข้อตัวแปลภาษา ปกติเราอ้างถึง เวลาผูกโยงของลักษณะประจำ คือ เวลาเร็วที่สุด (earliest time) ซึ่งกฎของภาษา อนุญาตให้มี การผูกโยงลักษณะประจำ

ตัวอย่างเวลาของการผูกโยง จงพิจารณา ตัวอย่างของลักษณะประจำก่อนหน้านี้ ในการประกาศ

```
const    n = 2;
```

ค่า 2 ผูกโยงอย่างคงที่ กับ ชื่อ n และในการประกาศ

```
var      x : integer;
```

ข้อมูลชนิด integer ผูกโยงอย่างคงที่ กับชื่อ x เช่นเดียวกันกับ ข้อความสั่ง ซึ่งเป็นการประกาศฟังก์ชัน

ในทางตรงกันข้าม การกำหนดค่า x := 2 ผูกโยงค่า 2 อย่างพลวัต กับ x เมื่อข้อความสั่งกำหนดค่า ถูกกระทำการ และ ข้อความสั่ง

```
new(y);
```

ผูกโยงอย่างพลวัต ตำแหน่ง หน่วยเก็บ ให้กับ y^ และกำหนด ให้เป็นตำแหน่งนี้ เป็นค่าของ y

เวลาของการผูกโยง อาจแบ่งให้ละเอียดยิ่งขึ้น เป็น ชนิดย่อยของการผูกโยงแบบพลวัต และ การผูกโยงแบบคงที่

ลักษณะประจำแบบคงที่ สามารถถูกผูกโยงระหว่างการวิเคราะห์กระจาย หรือวิเคราะห์ความหมาย (เวลาแปลโปรแกรม) ระหว่างการโยงของโปรแกรม กับ libraries (เวลาโยง) หรือ ระหว่างการบรรจุโปรแกรม สำหรับการกระทำการ (เวลาบรรจุ)

ตัวอย่างเช่น body ของ **ฟังก์ชันนิยามภายนอก** (an externally defined function)) จะไม่มีการผูกโยง จนกว่าจะถึง เวลาโยง (link time) และตำแหน่งของ ตัวแปรส่วนกลางของ Pascal จะผูกโยง ณ เวลาบรรจุ (at load time) เพราะว่า ตำแหน่งของมัน ไม่มีการเปลี่ยนแปลง ระหว่างการกระทำการของโปรแกรม

ชื่อต่างๆ (names) ผูกโยงให้กับ ลักษณะประจำ (attributes) ก่อนเวลาแปลโปรแกรม (translation time)

Redefined identifiers เช่น ข้อมูลชนิด boolean และ char ของ Pascal มีความหมายของ

มัน (และมีลักษณะประจำ) กำหนดโดย บทนิยามของภาษา

ตัวอย่างเช่น ข้อมูลชนิด boolean กำหนดให้มีค่าสองอย่าง คือ **true** หรือ **false**

predefined identifiers บางตัว เช่น ข้อมูลชนิด **integer** และตัวคงที่ **maxint** มีลักษณะประจำของมัน ซึ่งกำหนดโดยบทนิยามของภาษา และโดยการทำให้เกิดผล (by the implementation)

บทนิยามของภาษากำหนดว่า ข้อมูลชนิด **integer** มีค่าต่างๆ ซึ่งประกอบด้วยเซตย่อยของ จำนวนเต็ม และ maxint ซึ่งเป็นตัวคงที่ ขณะที่การทำให้เกิดผล กำหนดค่า ของ **maxint** และ พิสัยจริง (actual range) ของข้อมูลชนิด **integer**[1]

ดังนั้น เวลาผูกโยง ที่เป็นไปได้ สำหรับ ลักษณะประจำของชื่อ มีดังนี้

- Language definition time
- Language implementation time
- Translation time
- Link time
- Load time
- Execution time

เวลาผูกโยงทั้งหมด ในรายการข้างต้นนี้ เป็นการผูกโยงแบบคงที่ ยกเว้น เวลาผูกโยงสุดท้าย ซึ่งเป็น การผูกโยงแบบพลวัต

12-2

---

[1] สิ่งนี้ หมายความว่า ตัวโปรแกรม ไม่จำเป็นต้อง ให้ความหมายโดยละเอียด ของชื่อเหล่านี้ ในการประกาศ

การให้ บทนิยามใหม่ อาจเป็นไปได้ เนื่องจาก predefined identifiers เหล่านี้ ไม่ใช่ reserved words

Terrence W. Pratt[12] กล่าวว่า **การผูกโยง** ไม่ใช่แนวคิด ซึ่งจะให้บทนิยามที่แน่นอนได้หนึ่งอย่าง เช่นเดียวกับ **เวลาการผูกโยง**

(Binding is not **a** concept that **allows** a single precise definition, nor is **binding time.)**

การผูกโยง ใน ภาษาโปรแกรม มีต่างๆ กันมากมาย เช่นเดียวกับ ความหลากหลายของเวลาการผูกโยง เราอาจพูดถึง

การผูกโยง ของ สมาชิกของโปรแกรม กับ ลักษณะเฉพาะอย่างหนึ่ง หรือคุณสมบัติ อย่างหนึ่ง ซึ่งเลือกจาก เซตของคุณสมบัติที่เป็นไปได้

(We may speak of **the binding** of a program element to a paticular characteristics or property as simply the choice of the property from a set of possible properties.)

เวลาระหว่างการกำหนดโปรแกรม หรือ การประมวลผล เมื่อการเลือกนี้ ถูกกระทำ เรียกว่า **เวลาการผูกโยง** ของคุณสมบัติ สำหรับสมาชิกตัวนั้น

(The time during program formulation or processing when this choice is made is termed the **binding** of that property for that element.)

นอกจากนี้แล้ว ในแนวคิดของการผูกโยงและเวลาการผูกโยง เรายังรวมคุณสมบัติของสมาชิกของโปรแกรม ซึ่งคงที่ ไม่ว่าจะเป็นโดย บทนิยามของภาษา หรือ โดย การทำให้เกิดผลของภาษา

ชนิดของเวลาการผูกโยง (Classes of Binding Times)

ในขณะที่ยังไม่มี การจำแนกประเภทต่างๆ ของการผูกโยง เวลาการผูกโยงที่สำคัญ อาจแบ่งแยกได้ ถ้าเรายังจำได้ถึงข้อสมมติฐานเบื้องต้น ของการประมวลผลของ โปรแกรม ไม่ว่าจะเป็นภาษาใดๆ ก็ตาม ปกติจะเกี่ยวข้องกับ ขั้นการแปลภาษา (translation step) และตามด้วยขั้นการกระทำการ (execution) ของ โปรแกรมซึ่งถูกแปลแล้ว :

---

[12] Programming Languages : Design and Implementation **หน้า 30**

1) **เวลากระทำการ หรือ เวลาวิ่งโปรแกรม** (Execution time หรือ run time)

การผูกโยงจำนวนมาก ถูกกระทำระหว่างการกระทำการโปรแกรม การผูกโยงเหล่านี้ ได้แก่ การผูกโยงของตัวแปร กับค่าของมัน (bindings of variables to their values) เช่นเดียวกับในภาษาจำนวนมาก การผูกโยงของตัวแปร กับ ตำแหน่งหน่วยเก็บเฉพาะ (the binding of variables to particular storage locations) ชนิดย่อยที่สำคัญ สองอย่าง ซึ่งอาจแยกให้เห็น ได้แก่

(a) **การเข้าไปในโปรแกรมย่อย หรือบล็อก** (On entry to a subprogram or block) ในภาษาส่วนใหญ่ การผูกโยงชนิดที่สำคัญ ถูกจำกัด ให้เกิดขึ้นเฉพาะ ณ เวลา ของการเข้าไปยังโปรแกรมย่อย หรือ เข้าไปยังบล็อก ระหว่างการกระทำการ

ตัวอย่างเช่น ในภาษา Pascal การผูกโยงของ พารามิเตอร์ทางการ (formal parameters) กับ พารามิเตอร์จริง (actual parameters) และการผูกโยง ของ พารามิเตอร์ทางการ กับ ตำแหน่งหน่วยเก็บ อาจเกิดขึ้นได้ เฉพาะเมื่อเข้าไปยังโปรแกรมย่อยเท่านั้น

(b) **ณ จุดใดๆ ระหว่างการกระทำการ** (At arbitary point during execution)

การผูกโยงที่สำคัญชนิดอื่นๆ อาจเกิดขึ้น ณ เวลาใดๆ ระหว่างการกระทำการของโปรแกรม ตัวอย่างที่สำคัญมากที่สุดในที่นี้คือ การผูกโยงพื้นฐาน ของ ตัวแปร ให้กับค่าต่างๆ ระหว่างการกำหนดค่า

2) **เวลาแปลโปรแกรม หรือ เวลาคอมไพล์** (Translation time or compile time)

ภาษาโปรแกรมทั้งหมด โดยเฉพาะภาษาแปลความ (compiled languages) การผูกโยงชนิดที่สำคัญ ถูกกระทำ ระหว่างการแปลภาษา ซึ่งแบ่งออกเป็น สองชนิดที่แตกต่างกัน ดังนี้

(a) **การผูกโยงถูกเลือกโดยโปรแกรมเมอร์** (Bindings chosen by the programmer)

ในการเขียนโปรแกรม โปรแกรมเมอร์เป็นผู้กระทำการตัดสินใจ มากมาย เช่น เลือกชื่อตัวแปร ชนิดของตัวแปร โครงสร้างข้อความสั่งของโปรแกรม เป็นต้น ซึ่ง แทน (represent) การผูกโยงระหว่าง การแปลภาษา (translation) ตัวแปลภาษา จะใช้การผูกโยงเหล่านี้ ในการหา รูปแบบสุดท้าย ของ โปรแกรมภาษาจุดหมาย

(The language translator makes use of these binding in determining the final form of the object program.)

(b) **การผูกโยงถูกเลือกโดยตัวแปลภาษา** (Bindings chosen by the translator)

การผูกโยงอื่นๆ ถูกเลือกโดยตัวแปลภาษา โดยไม่มีการ ชี้นำ จาก ข้อกำหนดของ

โปรแกรมเมอร์ ตัวอย่างเช่น ในภาษา FORTRAN, การผูกโยงของตัวแปร กับ ตำแหน่งหน่วยเก็บ ถูกกระทำ ณ **เวลาบรรจุ** (load time) ซึ่งเป็นขั้นตอนสุดท้ายของการแปลภาษา เมื่อ โปรแกรมซึ่งถูกแปลแล้ว ถูก โยง และ บรรจุ เข้าไปยัง หน่วยความจำ ใน รูปแบบกระทำการได้สุดท้าย ของมัน (in their final executable form)

ภาษาโปรแกรมทั้งหมด การผูกโยง ของ โปรแกรมภาษาต้นฉบับ ให้กับ การแทนที่โปรแกรมภาษาจุดหมายใดๆ ถูกกระทำโดย ตัวแปลภาษา (In all languages, the binding of the source program to a particular object program representation is made by the translators.)

3) **เวลาทำภาษาให้เกิดผล** (Language implementation time)

บางด้าน ของ บทนิยามของภาษา อาจเหมือนกันหมดสำหรับทุกโปรแกรม ซึ่งวิ่ง (run) โดยใช้ การทำให้เกิดผล อย่างหนึ่ง ของภาษาหนึ่ง แต่บางด้านอาจจะผันแปร (vary) ได้ระหว่างการทำให้เกิดผลต่างๆ ตัวอย่างเช่น บ่อยครั้ง ซึ่งเป็นรายละเอียด ที่เกี่ยวข้องกับการแทนที่ของเลข (numbers) และ ของ การปฏิบัติการคำนวณ (arithmetic operations) ซึ่งหาได้จากวิธีซึ่งการคำนวณนั้น ถูกกระทำ ภายใต้ฮาร์ดแวร์ของคอมพิวเตอร์

โปรแกรมที่เขียน ในภาษา ซึ่งใช้ลักษณะของ บทนิยาม กำหนดคงที่ ณ เวลาทำให้เกิดผล ไม่จำเป็นต้องวิ่งบน การทำให้เกิดผล อีกอย่างหนึ่ง ของ ภาษาเดียวกัน เพราะว่า ปัญหาที่มีขึ้น มีมากกว่า มันอาจวิ่ง และให้ผลลัพธ์แตกต่างกัน

4) **เวลากำหนดบทนิยามภาษา** (Language definition time)

โครงสร้างส่วนใหญ่ของภาษาโปรแกรม จะคงที่ ณ เวลา ซึ่งให้นิยามภาษานั้น นั่นคือ ข้อกำหนดของทางเลือกต่างๆ ที่มีให้โปรแกรมเมอร์ ใช้เมื่อเขียนโปรแกรม (in the sense of specification of the alternatives available to a programmer when writing a program.) ตัวอย่างเช่น รูปแบบต่างๆ ของ ข้อความสั่งทางเลือกที่เป็นไปได้ ชนิดต่างๆ ของโครงสร้างข้อมูล โครงสร้างโปรแกรม เป็นต้น ทั้งหมดนี้ บ่อยครั้ง จะคงที่ ณ เวลาให้บทนิยามภาษา

เพื่อแสดงให้เห็น ความหลากหลายของการผูกโยง และเวลาการผูกโยง จงพิจารณา ข้อความสั่งกำหนดค่า ข้างล่างนี้

```
x := x + 10
```

สมมติว่า ข้อความสั่งนี้ เกิดขึ้นภายใน โปรแกรมที่เขียนด้วย ภาษา L เราอาจตั้งคำถามเกี่ยวกับ การผูกโยง และเวลาการผูกโยง อย่างน้อยที่สุด สมาชิกต่อไปนี้ ของ ข้อความสั่งข้างต้น

(a) **เซตของชนิดข้อมูลที่เป็นไปได้ของตัวแปร x** (Set of possible types for variable x)

ตัวแปร x ในข้อความสั่ง ปกติ จะมี แบบชนิดข้อมูลที่เกี่ยวข้องกับมัน (usually has a data type associated with it.) เช่น เป็น real, integer หรือ boolean

เซตของชนิดที่ยอมให้ใช้ได้ ของ x บ่อยครั้งจะคงที่ ณ เวลากำหนดบทนิยามภาษา ตัวอย่างเช่น อนุญาตให้เป็นได้เฉพาะ ข้อมูลชนิด real, integer, boolean, set และ character เท่านั้น

อีกทางเลือกหนึ่งคือ ภาษาโปรแกรมแบบนี้อาจยอมให้ แต่ละโปรแกรม นิยาม ข้อมูลชนิดใหม่ ได้เอง เช่น ใน Pascal และ Ada ดังนั้น เซตของแบบชนิดข้อมูลที่เป็นไปได้ สำหรับ x จะคงที่ ณ เวลาแปลโปรแกรม

(b) **ชนิดของตัวแปร x** (Type of variable x) ชนิดข้อมูลอย่างหนึ่ง ซึ่งเกี่ยวข้องกับ ตัวแปร x บ่อยครั้ง คงที่ ณ เวลาแปลโปรแกรม รวมทั้ง การประกาศชัดแจ้ง ในโปรแกรม เช่น ภาษา Pascal ดังนี้

```
var  x : real;
```

หมายถึง ข้อมูลชนิด real ถูกผูกโยงแบบคงที่ ให้กับ ชื่อ x

ในภาษาโปรแกรมอื่นๆ เช่น APL และ LISP ชนิดข้อมูลของ x อาจถูกผูกโยง เฉพาะ ณ เวลากระทำการเท่านั้น โดยผ่านการ กำหนดค่า ของ ชนิดอย่างหนึ่ง ที่ให้กับ x ในภาษาเหล่านี้ x อาจอ้างถึง แถวลำดับ ณ จุดหนึ่ง และอ้างถึง integer ณ อีกจุดหนึ่ง ต่อมา ใน โปรแกรมเดียวกัน

(c) **เซตของค่าที่เป็นไปได้ ของตัวแปร x** (Set of possible values for variable x)

ถ้า x มี ชนิดข้อมูลเป็น real ดังนั้น ค่าของมัน ณ เวลาใดๆ ระหว่าง การกระทำการ หมายถึง ค่าหนึ่ง ของเซต ของ การแทนที่ ลำดับบิต ของเลขจำนวนจริง (is one of set of bit seqences representing real numbers.)

เซตแน่นอน ของ ค่าเป็นไปได้ สำหรับ x หาได้จาก จำนวนจริง ซึ่ง สามารถถูกแทนที่ และ คุมแต่ง (manipulated) ในคอมพิวเตอร์เสมือน ซึ่งนิยามภาษานั้น ซึ่งปกติ หมายถึง เซต ของ จำนวนจริง ซึ่งสามารถถูกแทนที่ได้อย่างสะดวก ใน ฮาร์ดแวร์ของคอมพิวเตอร์ ดังนั้น เซตของ ค่าที่เป็นไปได้ของ x จะหาได้ ณ เวลาการทำภาษาให้เกิดผล การทำให้เกิดผลที่ต่างๆ กันของภาษา จะให้ พิสัยแตกต่างกันของค่าที่เป็นไปได้ของ x

(d) **ค่าของตัวแปร x** (Value of the variable x)

ณ จุดใดๆ ระหว่างการกระทำการโปรแกรม ค่าหนึ่งค่า จะถูกผูกโยง ให้กับตัวแปร x ปกติ ค่านี้ หาได้ ณ เวลากระทำการ ผ่านการกำหนดค่า หนึ่งค่า ให้กับ x ดังนั้น

- การกำหนดค่า x := x + 10 เปลี่ยนแปลงการผูกโยงของ x โดย แทนที่ค่าเก่าของมัน ด้วยค่าใหม่ ซึ่งมากกว่า ค่าเก่า อีก 10

(e) **การแทนที่ของตัวคงที่** 10 (Representation of the constant 10)

จำนวนเต็ม สิบ มีการแทนที่ เป็นค่าคงที่ ในโปรแกรม โดยใช้ string 10 และมีการแทนที่ ณ เวลากระทำการ เป็นลำดับของ บิต (a sequence of bits) ทั้งสองอย่าง

การเลือก การแทนที่ของเลขฐานสิบ ใน โปรแกรม (เช่น ใช้ 10 แทนเลขสิบ) ปกติ กระทำ ณ เวลากำหนดบทนิยาม ของภาษา ในขณะที่ การเลือกของลำดับอย่างหนึ่งของบิต เพื่อแทนเลขสิบ ณ เวลากระทำการ ปกติจะกระทำ ณ เวลาทำภาษาให้เกิดผล

(f) **คุณสมบัติของตัวปฏิบัติการ** + (Properties of the operator +) เมื่อพิจารณาเวลาผูกโยง ของคุณสมบัติต่างๆ ของ ตัวปฏิบัติการ + ในข้อความสั่ง การเลือกสัญลักษณ์ + เพื่อแทน การปฏิบัติการบวก กระทำ ณ เวลากำหนดบทนิยามภาษา อย่างไรก็ตาม สัญลักษณ์ + อย่างเดียวกันนี้ อนุญาตให้ใช้แทน real addition, integer addition, complex addition เป็นต้น ทั้งนี้ขึ้นอยู่กับ บริบท (context)

ในภาษาแปลความ จะทำการหาว่าการปฏิบัติการชุดไหน จะถูกแทนที่ด้วย + กระทำ ณ เวลาคอมไพล์ กลไก สำหรับการกำหนด การผูกโยงที่ต้องการ ปกติ คือ กลไกของชนิด สำหรับตัวแปร ดังนี้ :

ถ้า x เป็นชนิด integer แล้ว สัญลักษณ์ + ใน x + 10 แทน integer addition

ถ้า x เป็นชนิด real แล้ว สัญลักษณ์ + ใน x + 10 แทน real addition เป็นต้น

คุณสมบัติอีกอย่างหนึ่ง ของ การปฏิบัติการซึ่งแทนด้วย + คือ ค่าของมันสำหรับ คู่ที่กำหนดให้ใดๆ ของตัวถูกดำเนินการ ดังนั้น ในตัวอย่างของเรา ถ้า x มีค่าเป็น 12 แล้ว x + 10 มีค่าเป็นอะไร พูดอีกอย่างหนึ่งคือ ความหมายของ การบวกถูกนิยามเมื่อใด? (In other words, when is the meaning of addition defined?)

ตัวปฏิบัติการ ในภาษา Pascal หมายถึง สัญลักษณ์ หนึ่งอย่าง ประกอบด้วยตัวอักขระ

หนึ่งตัว หรือ กลุ่มของตัวอักขระ ซึ่งแทนการปฏิบัติการ ซึ่งจะถูกกระทำบนค่าหนึ่งค่า หรือ มากกว่าหนึ่งค่า ของชนิดข้อมูล ซึ่งกำหนดให้ สุดท้าย จะได้ผลลัพธ์ หนึ่งค่า และไม่จำเป็นต้องเป็นข้อมูลชนิดเดิม ความหมาย ปกติ คงที่ ณ เวลาทำภาษาให้เกิดผล และเอามาจากบทนิยามของการบวก ซึ่งใช้ ในคอมพิวเตอร์ฮาร์ดแวร์

(Opertor in Pascal, a symbol, consisting of a single character or group of characters, that represents an operation to be performed on one or more values of a given data type, culminating in a single result, not necessarily of the same data type,)

บทสรุป สำหรับภาษาเหมือน Pascal :

สัญลักษณ์ + ถูกผูกโยงให้กับ เซตของ การปฏิบัติการบวก ณ เวลากำหนดบทนิยามของภาษา (the symbol + is bound to a set of **addition opertions** at language definition time)

การปฏิบัติการบวกแต่ละชุด ในเซต ถูกนิยาม ณ เวลาทำภาษาให้เกิดผล (each addition operation in the set is defined at language implementation time)

การใช้สัญลักษณ์ + แต่ละครั้งในโปรแกรม ผูกโยงกับ การปฏิบัติการบวก อย่างหนึ่ง ณ เวลาแปลโปรแกรม (each particular use of the symbol + in a program is bound to a particular addition operation at translation time)

และค่าอย่างหนึ่ง ของ การปฏิบัติการบวก แต่ละชุด สำหรับตัวถูกดำเนินการของมัน หาได้เฉพาะ ณ เวลากระทำการเท่านั้น (and the particular value of each particular addition operation for its operands is determined only at execution time.)

เซตของการผูกโยงนี้ แทน การเลือกหนึ่งอย่างของ การผูกโยงที่เป็นไปได้ และเวลาผูกโยง ปกติ ของ ความหลากหลาย ของ ภาษาโปรแกรม (This set of bindings represents one choice of possible bindings and binding times typical languages.)

ข้อสังเกต อย่างไรก็ตาม การผูกโยงอื่นๆ และเวลาการผูกโยง เป็นสิ่งที่เป็นไปได้ ในภาษา SNOBOL4 การผูกโยงเหล่านี้ ทั้งหมด กระทำ ณ เวลากระทำการ

ความสำคัญของเวลาการผูกโยง (Important of Binding Times)

ความสำคัญส่วนใหญ่ และข้อแตกต่างปลีกย่อย ระหว่างภาษาโปรแกรมต่างๆ จำนวนมาก

เกี่ยวข้องกับ ความแตกต่าง ในเวลาผูกโยง ตัวอย่างเช่น **เกือบทุกภาษา** ยอมให้ เลข (number) เป็นข้อมูลและยอมให้ การปฏิบัติการคำนวณ กระทำบนเลขเหล่านี้

แต่ไม่ใช่ทุกภาษา มีความเหมาะสมเท่ากัน สำหรับปัญหาการเขียนโปรแกรม เกี่ยวข้องกับการคำนวณ จำนวนมาก ตัวอย่างเช่น ขณะที่ทั้งภาษา SNOBOL4 และ FORTRAN ยอมให้เรากำหนดและคุมแต่งแถวลำดับของตัวเลข การแก้ปัญหาของแถวลำดับขนาดใหญ่ และปริมาณของการคำนวณมากๆ ภาษา SNOBOL4 น่าจะไม่เหมาะสมมากที่สุด ถ้าหากงานนี้ สามารถกระทำได้ ในภาษา FORTRAN ถ้าเราพยายามที่จะตามรอย (trace) เหตุผลนี้ โดยการเปรียบเทียบคุณสมบัติของภาษา SNOBOL4 และ FORTRAN สุดท้าย จะสรุปได้ว่า ข้อดีที่สุด ของ FORTRAN ในกรณีนี้คือ ความจริงที่ว่า ใน SNOBOL4 ส่วนใหญ่ของการผูกโยงที่ต้องการ ในโปรแกรมจะเกิดขึ้น (set up) ณ เวลากระทำการ ในขณะที่ ภาษา FORTRAN ส่วนใหญ่ จะเกิดขึ้น ณ เวลาแปลโปรแกรม ดังนั้น โปรแกรมเวอร์ชัน SNOBOL4 จะใช้ เวลากระทำการของมันส่วนใหญ่ creating และ destroying การผูกโยง ในขณะที่โปรแกรมเวอร์ชัน FORTRAN การผูกโยงอย่างเดียวกัน ส่วนใหญ่ เกิดขึ้นครั้งเดียว ระหว่าง การแปลโปรแกรม และเหลือเพียงส่วนน้อยเท่านั้น ซึ่งจะถูกจัดกระทำ (handled) ระหว่างการกระทำการ

ผลลัพธ์คือ โปรแกรมเวอร์ชัน FORTRAN จะกระทำการได้อย่างมีประสิทธิภาพมากกว่า ในทางตรงกันข้าม เราอาจจะย้อนกลับมา ถามคำถามที่เกี่ยวข้องกันคือ "ทำไม FORTRAN จึงไม่คล่องตัว ในการจัดกระทำ ของแถวลำดับ เลข และ การคำนวณ เมื่อเปรียบเทียบกับ SNOBOL4"

("Why is FORTRAN so inflexible in its handling of arrays, numbers, and arithmetic, as compared to SNOBOL4?")

<u>คำตอบ</u> เกี่ยวข้องกับเวลาผูกโยง เพราะว่าการผูกโยงส่วนใหญ่ ในภาษา FORTRAN กระทำ ณ เวลาแปลโปรแกรม ก่อนที่จะทราบค่าข้อมูล อินพุท (before the input data are known) จึงเป็นการยาก ในภาษา FORTRAN ที่จะเขียนโปรแกรม ซึ่งสามารถ ดัดแปลง (adapt) ให้กับความหลากหลายของสถานการณ์ ที่ขึ้นอยู่กับข้อมูลต่างๆ ณ เวลากระทำการ ตัวอย่างเช่น ภาษา FORTRAN ขนาดของแถวลำดับ และชนิดของตัวแปร ต้องคงที่ ณ เวลาแปลโปรแกรม ส่วนการผูกโยง ในภาษา SNOBOL4 อาจจะถูกทำให้ช้าออกไป (delayed) ระหว่าง การกระทำการ จน

กระทั่ง ข้อมูลอินพุท ได้มีการตรวจสอบ และทำการผูกโยงอย่างเหมาะสม สำหรับ ข้อมูลอินพุท ที่หาได้

ภาษาโปรแกรม เหมือน FORTRAN ซึ่งการผูกโยง ส่วนใหญ่ กระทำ ระหว่างการแปลโปรแกรม **ทำก่อน** การประมวลผลของโปรแกรม เรียกว่า **มีการผูกโยงเร็ว** (early binding) ส่วนภาษาที่เรียกว่า **การผูกโยงช้า** (late bindings) เช่น SNOBOL4 การผูกโยงส่วนใหญ่ช้าออกไปจนถึงเวลากระทำการ

ข้อดีและข้อไม่ดี ของ การผูกโยงเร็ว และการผูกโยงช้า มีข้อขัดแย้งกัน ระหว่าง ประสิทธิภาพ และความคล่องตัว

ภาษา ซึ่งสิ่งแรกที่ควรพิจารณา คือ ประสิทธิภาพในแง่การกระทำการ (execution efficiency) เช่น FORTRAN, Pascal และ COBOL เป็นภาษาซึ่งออกแบบมาเพื่อให้การผูกโยงจำนวนมากเท่าที่เป็นไปได้ ถูกกระทำ ระหว่างการแปลโปรแกรม

ในขณะที่สิ่งแรกของการพิจารณาคือ ความคล่องตัว (flexibility) เช่นภาษา SNOBOL4 และ LISP การผูกโยงส่วนใหญ่ ถูกทำให้ช้าออกไป จนกระทั่งถึงเวลากระทำการ เพื่อให้มันกระทำกับ data-dependent

ส่วนภาษา ซึ่งออกแบบมาสำหรับ การกระทำการที่มีประสิทธิภาพ และมีความคล่องตัวทั้งสองอย่าง เช่น Ada และ PI/1 มีทางเลือกหลายอย่างให้ใช้ได้เพื่อเลือกเวลาผูกโยง (multiple options are often available that allow choices of binding times.)

แบบฝึกหัด

1. Write a statement in a language with which you are familiar. For each syntactic component of the statement (variable names, operation symbols, etc.), list the various bindings that are necessary to completely determine the semantics of the statement when it is executed. For each binding, identify the binding time used in the language.
2. Do Problem 1, but **use a declaration** instead of a statement. Declarations are usually said **to be elaborated** instead of executed, so list the bindings (and their binding times) necessary for a complete elaboration of the declaration.)

การผูกโยง ต้องถูกรักษาไว้ (maintained) โดยตัวแปลภาษา (translator) เพื่อความหมายที่เหมาะสม จะถูกกำหนดให้กับ ชื่อต่างๆ ระหว่างการแปลโปรแกรม และการกระทำการ ตัวแปลภาษา ทำสิ่งนี้ โดยการสร้าง (creating) โครงสร้างข้อมูลขึ้นหนึ่งชุด เก็บ สารสนเทศ เนื่องจากว่าเราไม่สนใจในรายละเอียด ของโครงสร้างข้อมูลนี้ แต่สนใจเฉพาะคุณสมบัติของมัน เราสามารถคิดอย่างนามธรรมว่า โครงสร้างข้อมูลนี้ เป็นฟังก์ชัน แสดงให้เห็นการผูกโยง ของลักษณะประจำกับ ชื่อ

ฟังก์ชันนี้ เป็น ส่วนหนึ่งที่สำคัญ ของอรรถศาสตร์ ของภาษา ปกติ เรียกว่า **ตารางสัญลักษณ์**

(This function is a fundamental part of language semantics and is usually called the **symbol table**.)

ในเชิงคณิตศาสตร์ ตารางสัญลักษณ์ หมายถึง ฟังก์ชัน จาก ชื่อ ไปยัง ลักษณะประจำ ซึ่งเขียนดังนี้

Symbol Table : Names $\longrightarrow$ Attributes

หรือเขียนเป็นรูปภาพดังนี้

Symbol Table

Names $\longrightarrow$ Attributes

ฟังก์ชันนี้ จะเปลี่ยนแปลง ขณะดำเนินการแปลโปรแกรม และ/หรือ กระทำการโปรแกรม เพื่อสะท้อน การใส่ (additions) และการลบทิ้ง (deletions) ของ การผูกโยงต่างๆ ภายในโปรแกรม ขณะ แปล และ/หรือ กระทำการ ตารางสัญลักษณ์ จะศึกษารายละเอียดมากขึ้น ในอีก สองหัวข้อถัดไป

ข้อแตกต่างหลัก ที่มีอยู่ ระหว่าง วิธี ซึ่งตารางสัญลักษณ์ ถูกรักษาไว้โดยตัวแปลคำสั่ง กับ วิธี ซึ่ง ตารางสัญลักษณ์ ถูกรักษาไว้โดยตัวแปลภาษา เป็นดังนี้

จากบทนิยาม ตัวแปลโปรแกรม (compiler) คำนวณเฉพาะ ลักษณะประจำแบบคงที่ เนื่องจากโปรแกรม จะไม่กระทำการจนกว่า การแปล เสร็จสิ้น ดังนั้น ตารางสัญลักษณ์ สำหรับตัวแปลภาษา เขียนดังนี้

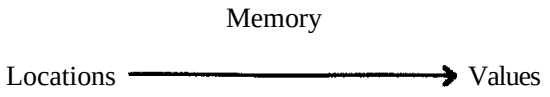
Symbol Table

Names $\longrightarrow$ Static Attributes

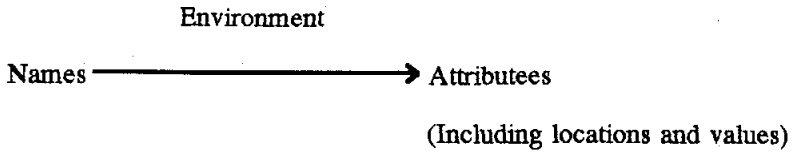
ระหว่างการกระทำการของ โปรแกรม ซึ่งแปลแล้ว (compiled program) ลักษณะประจำ เช่น ตำแหน่ง (locations) และค่าต่างๆ (values) ต้องถูกเก็บไว้ ตัวแปลโปรแกรม สร้าง (generate) รหัส ซึ่ง รักษา ลักษณะประจำเหล่านี้ ใน โครงสร้างข้อมูล (in data structures) ระหว่างการกระทำการ

การจัดสรรหน่วยความจำ (memory allocation) ของ กรรมวิธีนี้ คือ การผูกโยง ชื่อต่างๆ กับตำแหน่งหน่วยเก็บ ซึ่งปกติจะพิจารณาแยกต่างหาก เรียกว่า **สิ่งแวดล้อม** (environment) เขียนภาพ ดังนี้

สุดท้าย การผูกโยงของ ตำแหน่งหน่วยเก็บ (storage locations) กับค่าต่างๆ (values) เรียกว่า **หน่วยความจำ** (memory) เนื่องจาก มัน นามธรรม หน่วยความจำ ของ คอมพิวเตอร์ จริง (บางครั้งเรียกว่า store หรือ state) เขียนภาพดังนี้



ในทางตรงกันข้าม ในตัวแปลคำสั่ง ตารางสัญลักษณ์ และสิ่งแวดล้อม รวมเข้าด้วยกัน เพราะว่า ลักษณะประจำแบบคงที่ และแบบพลวัต ทั้งคู่ คำนวณระหว่าง การกระทำการ (execution) โดยปกติ หน่วยความจำ รวมฟังก์ชันนี้ด้วย เขียนภาพดังนี้



5.2 การประกาศ บล็อก และ สโคป

(Declarations, blocks, and scope)

การประกาศ หมายถึง วิธีสำคัญวิธีหนึ่ง สำหรับสร้างการผูกโยง (Declarations are a principal method for establishing bindings.)

การประกาศ อาจทำ**ชัดแจ้ง** (explicit) เช่นในภาษา Pascal และภาษาโปรแกรมอื่นๆ อีกจำนวนมาก หรือการประกาศ อาจทำ**โดยนัย** (implicit) พูดง่ายๆ คือ การใช้ชื่อของตัวแปร ทำให้มันถูกประกาศ

ภาษาซึ่งมีการประกาศโดยนัย ปกติ จะมี**ข้อตกลงของชื่อ** (name conventions) เพื่อสร้างลักษณะประจำอื่นๆ ตัวอย่างเช่น ภาษา FORTRAN ไม่ต้องมีการประกาศตัวแปร สำหรับ simple variables ตัวแปรทั้งหมด ใน FORTRAN ซึ่งไม่มีการประกาศชัดแจ้ง จะถือว่าเป็น จำนวนเต็ม (integer) ถ้าชื่อนั้นขึ้นต้นด้วย อักษร I, J, K, L, M หรือ N ส่วนกรณีอื่นๆ จะเป็นจำนวนจริง (real)

ข้อตกลงนี้ คล้ายกับ ข้อตกลงในภาษา BASIC บางเวอร์ชัน กล่าวคือ ตัวแปรใดๆ ก็ตาม ที่ลงท้ายด้วย "%" หมายถึง integer, ตัวแปรใดๆ ก็ตาม ที่ลงท้ายด้วย "$" หมายถึง string ส่วนกรณีอื่นๆ ทั้งหมด เป็น real ส่วนภาษาอื่นๆ ซึ่งมีการประกาศโดยนัย ได้แก่ภาษา APL และ SNOBOL

การประกาศ ปกติเกี่ยวข้องกับ โครงสร้างภาษาอย่างหนึ่ง ซึ่งเรียกว่า **บล็อก** (Declarations are usually associated with a particular language structure called a **block**.)

ภาษา Pascal แบ่งบล็อกออกเป็น สองชนิดคือ

- บล็อกของโปรแกรมหลัก (main program block) และ
- บล็อกของโปรแกรมย่อย (procedure/function block)

ตำแหน่งของการประกาศ ในบล็อกเหล่านี้ แสดงให้เห็นในภาพข้างล่างนี้

```
program ex;
  . . .
  procedure p;
    .        } declarations
             }   of p
  begin                          } declarations
    .                                of ex
  end;    (* p *)
begin (* main *)
  . . .
end. (* ex *)
```

การประกาศที่เกี่ยวข้องกับ e x เรียกว่า **การประกาศส่วนกลาง** ส่วนการประกาศที่เกี่ยวข้องกับ p เรียกว่า **การประกาศเฉพาะที่** ให้กับ p

(Declarations associated to **ex are called global declarations,** while declarations associated to p **are declaration local to** p.)

ในภาษา Algol60 และ Algo168, บล็อกประกอบด้วย ลำดับของข้อความสั่ง ปิดล้อมด้วยคู่ของ begin-end* การประกาศ จะอยู่หลัง begin แต่อยู่ก่อน **ข้อความสั่งแรก** (the first statement) ในบล็อก

ตัวอย่าง บล็อกของภาษา Algol60

```
begin
    integer x;
    boolean y;      } declarations
    x := 2;
    y := false;
    x := x + 1;
    ' '
end:
```

---

* ในภาษา Pascal ข้อความสั่งต่างๆ ซึ่งปิดล้อมด้วยคู่ begin-end เรียกว่า compound statements ไม่ใช่ blocks เพราะว่าการประกาศ ไม่ได้เกี่ยวข้องกับ ข้อความสั่งเหล่านี้

รูปแบบทั่วไป ของ บล็อก มีอยู่เช่นกันในภาษา C ซึ่งบล็อก ถูกปิดล้อมด้วยวงเล็บปีกกา

ตัวอย่าง ภาษา C

```
void  p(void)
{double  r, z;  /* the block of p */

    {int x, y;  /* another block */
        x := 2;
        y := 0;
        x += 1;
    }
    . . .
}
```

(**คำหลัก void** ในบทนิยาม void p(void) แสดงว่า บล็อก p ไม่มีค่าส่งกลับ และไม่มีพารามิเตอร์)

ภาษา C มี**สโคปภายนอก** (external scope) ซึ่งไม่ผูกติดกับ บล็อกใดๆ แต่สิ่งนั้น ทำให้เกิดสโคป ซึ่ง รวมฟังก์ชันทั้งหมดใน โปรแกรม (but that provides a scope that encloses all functions in a program.) ฟังก์ชันสโคปภายนอกนี้ เป็นสโคปส่วนกลาง (global scope) เพราะว่า โปรแกรมหลัก ในภาษา C เป็นเพียงอีกหนึ่งฟังก์ชัน และมีสโคปเฉพาะที่ของมันเอง แยกต่างหากจาก ส่วนที่เหลือของโปรแกรม

ตัวอย่าง ภาษา C

```
int  x;
float  y;
/* these are external to all functions and so global */
void  main(void)
{int i,j; /* these are local to main */
    . . .
}
```

ภาษา Ada รวม (combines) การประกาศ โปรซีเดอร์ และฟังก์ชันของ Pascal กับ บล็อก

ของภาษา เหมือน Algol เข้าด้วยกัน ยกเว้นที่ไม่เหมือน Algol60 คือ การประกาศ กำหนดโดยคำสงวน declare และอยู่ก่อนหน้าคู่ begin-end

ตัวอย่าง ภาษา **Ada**

```
declare x : INTEGER;
        y : BOOLEAN;
begin
    x := 2;
    y := 0;
    x := x + 1;
end;
```

โปรแกรมสำเร็จ และงานของ Ada ประกอบด้วย การประกาศ และรูปแบบของบล็อก

ภาษา Modula-2, บล็อก หมายถึง มอดูล (modules) และโปรซีเดอร์ (procedurs)

ส่วนภาษา FORTRAN, บล็อก หมายถึง โปรแกรมซับรูทีน (subroutine) และ ฟังก์ชัน (function)

การประกาศ ผูกโยง ลักษณะประจำหลายอย่างให้กับชื่อ ขึ้นอยู่กับชนิดของการประกาศ (Declarations bind various attributes to names, depending on the kind of declaration.) (ในหัวข้อที่แล้ว เราได้เห็นตัวอย่าง ของ ลักษณะประจำต่างๆ ซึ่ง ผูกโยง โดยการประกาศ

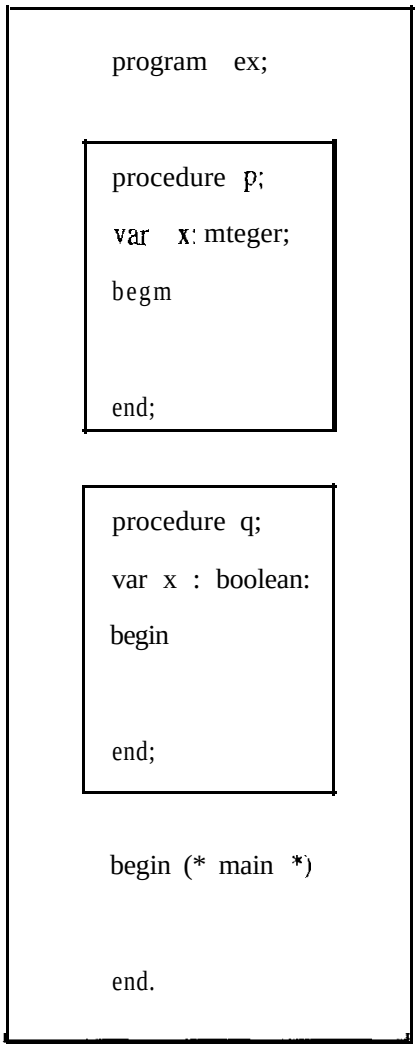
การประกาศ โดยตัวมันเอง มีลักษณะประจำอย่างหนึ่ง ซึ่งบอกได้โดยตำแหน่งของมันในโปรแกรม (A declaration itself has an attribute that is determined by its position in the program.)

สโคปของการประกาศ หมายถึง พื้นที่ของโปรแกรม ไปจนถึงที่ซึ่ง การสร้างการผูกโยงโดยการประกาศนั้น ถูกเก็บไว้)

(The **scope of a declaration** is the region of the program over which the bindings established by the declaration are maintained.)

เราสามารถ อ้างถึง สโคป ของการผูกโยง ตัวมันเองได้ บางครั้ง มีการใช้ผิด เราอ้างถึงสโคปของชื่อ สิ่งนี้อันตราย เพราะว่า ชื่อ เหมือนกัน อาจถูกเกี่ยวข้อง กับการประกาศแตกต่างกัน หลายชุด และการประกาศแต่ละชุด มี สโคปต่างกัน

ตัวอย่าง โปรแกรม ภาษา Pascal มีการประกาศ ชื่อ x สองครั้ง ที่มีความหมายต่างกัน และสโคปต่างกัน



ในภาษาเหมือน Pascal หรือ C ซึ่งบล็อกอาจอยู่ซ้อนกันได้ (nested) (นั่นคือ บล็อกหนึ่งอยู่ภายใน อีกบล็อกหนึ่ง) สโคปของการประกาศ ถูกจำกัดให้กับบล็อก ซึ่งมันปรากฏอยู่ (และบล็อกอื่นๆ ซึ่งอยู่ภายใน บล็อกนั้น) ภาษาเช่นนี้ เรียกว่า **ภาษาโครงสร้างแบบบล็อก** (block structured language) และชนิด กฎของสโคป สำหรับบล็อก ซึ่งอธิบายขณะนี้ เรียกว่า **สโคปเชิงศัพท์** (lexical scope) เพราะว่า มันเป็นไปตาม โครงสร้างของบล็อก ที่มันปรากฏ ในการเขียนรหัส มันเป็นกฎมาตรฐานในภาษาโปรแกรมส่วนใหญ่ (การอภิปราย เรื่องกฎสโคปที่แตกต่าง ให้ดูในหัวข้อถัดไป)

ตัวอย่าง สโคป ในภาษา Pascal

```
program ex;
var x : integer;

    procedure p;
    var y : boolean
    begin
        . . .
    end;    (* p *)

    procedure q;
    var z : real;
    begin
        . . .
    end; (* q *)

begin (* main of ex *)
    .
end,    (* ex *)
```

ใน โปรแกรมนี้ การประกาศของตัวแปร x โปรซีเดอร์ p และ โปรซีเดอร์ q เกี่ยวข้องกับ บล็อก ของ โปรแกรมหลัก ดังนั้น ทั้งหมดนี้ เป็น การประกาศส่วนกลาง (global declarations)

ในทางตรงกันข้าม การประกาศของ ตัวแปร y และ การประกาศของ ตัวแปร z เกี่ยวข้องกับ บล็อกของ โปรซีเดอร์ p และ โปรซีเดอร์ q ตามลำดับ จึงเป็นการประกาศ**เฉพาะที่** (local) กับ ฟังก์ชันเหล่านี้ และการประกาศ ของมัน ถูกต้อง (valid) เฉพาะ สำหรับ p และ q ตามลำดับ

ภาษา Pascal มีกฎว่า สโคปของการประกาศ เริ่มตั้งแต่ จุดของการประกาศ ตัวมันเอง เรียกว่า **กฎ การประกาศก่อนใช้** (declaration before use rule) ดังนั้น เราสามารถกล่าวกฎสโคป พื้นฐาน ของภาษา Pascal ดังนี้

สโคปของการประกาศ ขยาย จากจุด หลังการประกาศ ไปจนจบ don ซึ่งมันอยู่ (: the scope of a declaration extends from the point just after the declaration to the end of the block in which it is located.

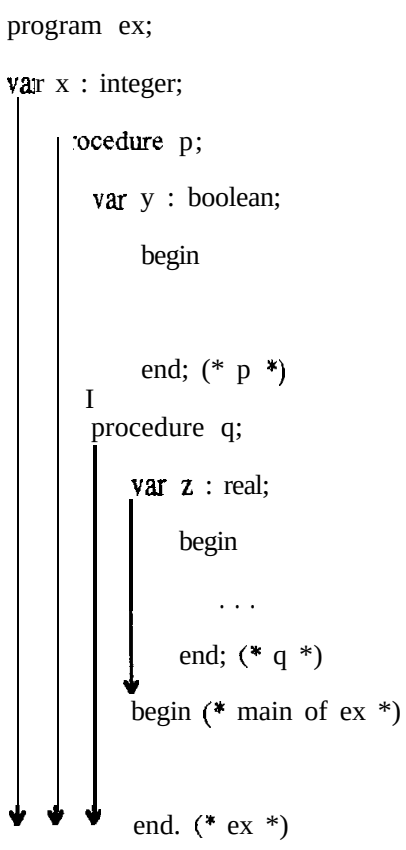
จากตำรา "Turbo pascal" หน้า 525

Scope in Pascal, the scope of an identifier refers to the set of blocks in a program in which the identifier can be used.

Rules of scope :

1. The scope of an identifier is the block in which it is declared. including all blocks nested within that block.
2. An identifier may be referenced only within its scope.
3. If an identifier is declared in block B nested in block A, then blockB and all blocks nested in it are excluded from the scope of the identifier's declaration in block A.

ตัวอย่าง ภาษา Pascal ชุดเดิม วาดลูกศร แสดงสโคปของการประกาศ แต่ละชุด



บล็อก ในภาษาเหมือน Algol สร้าง กฎสโคป อย่างเดียวกัน

ตัวอย่าง โปรแกรมภาษา Algol60

```
A : begin
        integer x;
        boolean y;
        x := 2;
        y := false;
    B : begin
            integer a, b;
            if y then a:=x
            else b := y;
        end;
        ' ' '
    end;
```

ตัวอย่างข้างต้นนี้ มี สองบล็อก คือ บล็อก A และ บล็อก B

สโคปของการประกาศ ของ x และ y คือ ทั้งหมดของบล็อก A (รวม บล็อก B ด้วย) ในขณะที่สโคปของ การประกาศของ a และ b คือบล็อก B เท่านั้น

ภาษา Modula-2 มี โครงสร้างบล็อก และกฎของสโคป เหมือนกับภาษา Pascal แต่มีข้อแตกต่างหลักอยู่สองอย่างคือ **ข้อแรก** สโคปของตัวแปร หรือ สโคปของการประกาศฟังก์ชัน ขยายจาก จุดเริ่มต้นของบล็อก ซึ่งมันอยู่ ไม่ใช่ จากจุดของการประกาศ ดังนั้น ในภาษา Modula-2 ไม่ติดกับกฎของการประกาศ ก่อนการใช้

ภาษา Modula-2 อนุญาตให้ การประกาศของ ตัวคงที่ ตัวแปร และโปรซีเดอร์ เรียงลำดับอย่างไรก็ได้ ไม่เหมือนกับ การเรียงลำดับที่เข้มงวดของภาษา Pascal มาตรฐาน (สิ่งแรก การประกาศ constants จากนั้น type, variable และ procedure ตามลำดับ)

ตัวอย่าง

```
MODULE Ex;
PROCEDURE p;
BEGIN
        x := 2; (* legal - global x defined below *)
END p;
VAR x : INTEGER:
BEGIN (* Ex *)

END Ex.
```

สโคปของตัวแปรส่วนกลาง x ขยายย้อนกลับ (backward) จากการประกาศของมัน ไปยังจุดเริ่มต้น ของ มอดูล Ex

ข้อแตกต่างประการที่สอง ในเรื่องกฎสโคป ระหว่างภาษา Modula-2 และภาษา Pascal คือ การใช้มอดูลเฉพาะที่ ใน Modula-2 กับ สโคปที่จำกัด จงพิจารณา ตัวอย่างข้างล่างนี้

```
MODULE A;
VAR x : INTEGER;
        PROCEDURE p;
        BEGIN
            . . .
        END p;
        MODULE B;
        VAR y : REAL;
                PROCEDURE q;
                BEGIN
                    . . .
                END q;
        E!ND B;
    . .
END A.
```

การประกาศ ของ ตัวแปร x และ โปรซีเดอร์ p มีสโคป คลุม MODULE A แต่ไม่รวม MODULE B นั่นคือ การอ้างถึง x และ p ภายใน B เป็นสิ่งไม่ถูกต้อง (illegal) ในทำนองเดียวกัน การอ้างถึง y และ q นอกมอดูล B เป็นสิ่งไม่ถูกต้องเช่นเดียวกัน มอดูลเฉพาะที่ (local modules) เช่น B ใน ฟังก์ชันตัวอย่างนี้ คือ **เขตแดนสโคป** (scope boundaries) หมายถึง ไม่มีสโคปคาบเกี่ยว เขตแดนเหล่านี้ ด้วยข้อยกเว้น ของการนำออกชัดแจ้ง หรือ การนำเข้าชัดแจ้ง (explicit exported or imported)

ตัวอย่างของการนำออกและการนำเข้า กำหนดให้ในรหัส ซึ่งมีการปรับปรุงข้างล่างนี้ เมื่อ โปรซีเดอร์ p ถูกนำเข้าไปใน มอดูล B และขณะนี้ สามารถถูกเรียกได้จาก B ในทำนองเดียวกัน ตัวแปร y ถูกนำออกไปจาก B และสามารถถูกอ้างถึงได้ใน A :

```
MODULE A;
VAR x : INTEGER;

PROCEDURE p;
BEGIN
  . .
END p;

    MODULE B;
    IMPORT p; (* p now available in B *)
    EXPORT y; (* y now available in A *)
    var y : REAL;

    PROCEDURE q;
    BEGIN
      . .
    END q;
    END B;
```

```
BEGIN

END A
```

สโคป ในภาษา FORTRAN สามารถ ขยาย ไปจนคลุม ซับรูทีน หรือ ฟังก์ชัน หลายชุด ได้ค่อนข้างคล้าย กับ วิธีนำออกของ Modula-2 โดยใช้ การประกาศ COMMON

ตัวอย่าง จงพิจารณา โปรแกรมภาษา FORTRAN ข้างล่างนี้

```
C     MAIN PROGRAM
      COMMON A
      A = 2.0
      B = 3.1
      CALL P
      PRINT 20, A, B
   20 FORMAT(2F3.1)
      END

      SUBROUTINE P
      COMMON A
      B = A
      A = A + 1.0
      RETURN
      END
```

ในโปรแกรมนี้ โปรแกรมหลัก และซับรูทีน P มีสโคปแตกต่างกัน ดังนั้น ตัวแปร B ในโปรแกรมหลัก ไม่ใช่ตัวเดียวกับ ตัวแปร B ในซับรูทีน P

ในทางตรงกันข้าม ข้อความสั่งCOMMON สองชุด สร้างตัวแปร สองตัว ชื่อ A ให้เป็นสิ่งเดียวกัน ดังนั้น A ใน P มีค่าเหมือนกับ A ใน โปรแกรมหลัก ดังนั้น เมื่อจบโปรแกรม B ยังคงมีค่าเท่ากับ 3.1 ในขณะที่ A มีค่าเท่ากับ 3.0 การใช้ COMMON ในภาษา FORTRAN สร้าง

สิ่งซึ่งเป็น ตัวแปรส่วนกลาง อย่างไรก็ตาม มันทำงานได้โดย การระบุตำแหน่ง ของ ตัวแปรต่างๆ ซึ่งมีรายชื่อใน ข้อความสั่ง COMMON และสิ่งนี้ มีผลกระทบไม่ปกติ (ดูหัวข้อ 5.6)

คุณสมบัติอย่างหนึ่งของ โครงสร้างบล็อก คือ การประกาศ ใน บล็อกซ้อนใน มีอันดับการทำก่อน สูงกว่าการประกาศก่อนหน้านั้น (One feature of block structure is that declarations in nested blocks take precedence over previous declarations.)

ตัวอย่าง จงพิจารณาโปรแกรมข้างล่างนี้

```
program ex:
var   x : integer;

    procedure p:
    var   x : boolean;
    begin
          x := true; (* x p *)

    end;

    begin
          x := 2:  (* global x *)

    end.
```

การประกาศของ x ในโปรซีเดอร์ p มีอันดับ การทำก่อน สูงกว่า การประกาศส่วนกลางของ x สำหรับช่วงเวลา ของ p ดังนั้น integer x ส่วนกลาง จึงไม่สามารถเข้าถึงได้ภายใน p การประกาศ ส่วนกลางของ x เรียกว่า มี **scope hole** ภายใน p สำหรับเหตุผลนี้ ความแตกต่าง คือ บางครั้ง ทำขึ้นระหว่าง สโคป (scope) และ การมองเห็นได้ (visibility) ของการประกาศ ดังนี้ :

การมองเห็นได้ ได้แก่ พื้นที่ต่างๆ ของโปรแกรม เฉพาะที่ซึ่งการผูกโยง ของ การประกาศ ถูกนำไปใช้ เท่านั้น (visibility includes only those regions of a program where the bindings of a declaration apply) ในขณะที่ สโคปนั้น รวม scope holes ด้วย (เพราะว่า การผูกโยง ยังคงมีอยู่ แต่ถูกซ่อนไม่ให้เห็น) ภาษา Ada **การประกาศซ่อน** (hidden declarations) เช่นนี้ ยังคงถูกเข้าถึงได้ โดยการใช้ scope qualifier เหมือนกับ record reference ดังนั้น ในภาษา Ada เราสามารถอ้างถึง x ส่วนกลาง ภายใน p ได้ โดยเขียน ex.x สิ่งนี้เรียกว่า **มองเห็นโดยการเลือก** (visibility by selection) ใน Ada

การผูกโยง ซึ่งสร้างโดย การประกาศ จะถูกเก็บไว้ โดยตารางสัญลักษณ์ (The bindings established by declarations are maintained by the symbol table.) วิธีซึ่งตารางสัญลักษณ์ ประมวลผล การประกาศ บอกได้จาก สโคป ของการประกาศ แต่ละชุด (The way the symbol table processes declarations determines the scope of each declaration.) ในหัวข้อถัดไป เราจะตรวจดู วิธีซึ่งตารางสัญลักษณ์ สามารถเก็บ สโคป ใน ภาษาโครงสร้างแบบบล็อก

5.3 ตารางสัญลักษณ์ (The Symbol Table)

ตารางสัญลักษณ์ ถูกเก็บไว้ โดย โครงสร้างข้อมูลจำนวนหนึ่ง เพื่อให้การเข้าถึง และ การบำรุงรักษา ตารางของชื่อ และลักษณะประจำ ทำได้อย่างมีประสิทธิภาพ (A symbol table can be maintained by any number of data structures to allow for efficient access and maintenance of a table of names and attributes) : ตารางแบบแฮช, ต้นไม้ และรายการ เป็น โครงสร้างข้อมูล บางอย่าง ซึ่งนำมาใช้ อย่างไรก็ตาม การบำรุงรักษา สารสนเทศของสโคป ใน lexically scoped language ด้วยโครงสร้างบล็อก ต้องกระทำดังนี้ การประกาศ ถูกประมวลผล ในลักษณะคล้ายกองซ้อน : การเข้าไปในบล็อก (on entry into a block) การประกาศทั้งหมด ของ บล็อกนั้น จะถูกประมวลผล และการผูกโยงที่สมนัยกัน ใส่เข้าไปในตารางสัญลักษณ์ หลังจากนั้น การออกจากบล็อก (on exit from the block) การผูกโยง ซึ่งจัดโดยการประกาศ ถูกลบออกด้วย การผูกโยงก่อนหน้านั้น ยังคงมีอยู่ ตารางสัญลักษณ์ไม่มีข้อจำกัดกับ โครงสร้างข้อมูลชนิดใดๆ ดังนั้นเราอาจมอง โครงร่างตารางสัญลักษณ์ เป็น การรวบรวมชื่อต่างๆ แต่ละชื่อ มี กองซ้อน ของการประกาศ หนึ่งชุด ที่เกี่ยวข้องกับ ชื่อนั้น เพื่อที่การประกาศ บนสุด ของกองซ้อน คือการประกาศ ซึ่งสโคปของมัน คือ ชุดที่ ถูกใช้งานปัจจุบัน

(We may nevertheless view the symbol table schematically as a collection of names,

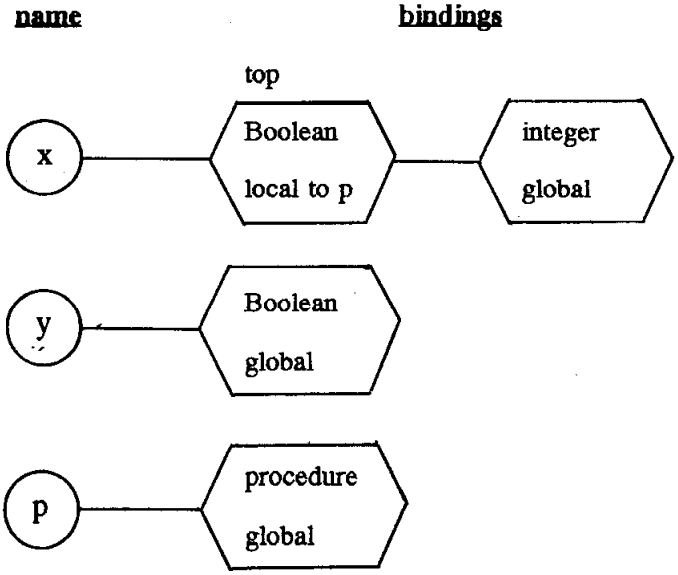
each of which has a stack of declarations associated to it, such that the declaration on top of the stack is the one whose scope is currently active.)

ตัวอย่าง จงพิจารณา โปรแกรมภาษา Pascal ต่อไปนี้

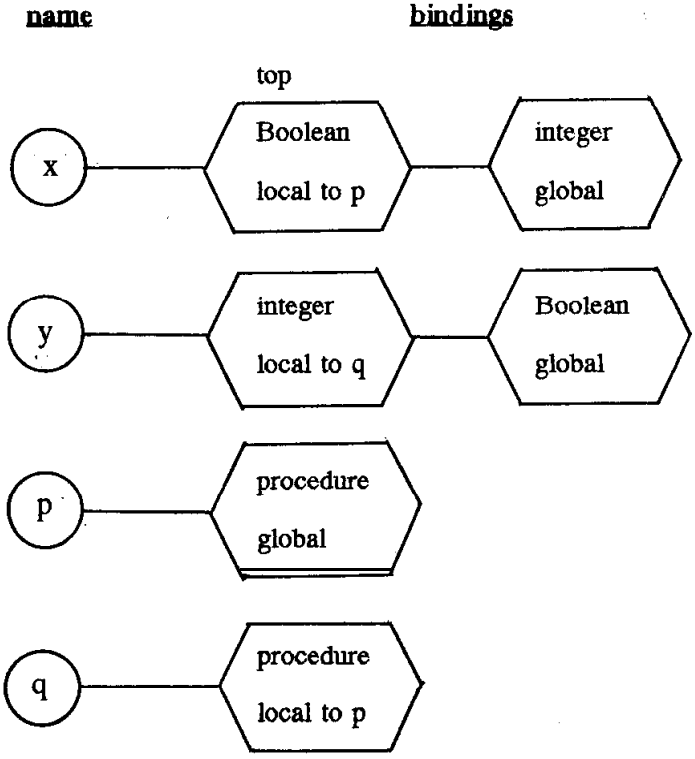
```
program  symtabex;
var  x  :  integer;
     y : boolean;

     procedure  p;
     var  x  :  boolean;

          procedure  q;
          var  y  :  integer;
          begin
               .
          end;  (* q *)

     begin  (* p *)
          .
     end;  (* p  *)

begin  (* main  *)
     . .
end.
```

ในที่นี้ x, y, p และ q เป็นชื่อ (names) ใน โปรแกรม symtabex แต่ x และ y เกี่ยวข้องกับ การประกาศ ที่แตกต่างกัน สองชุด ด้วย สโคปที่ต่างกัน

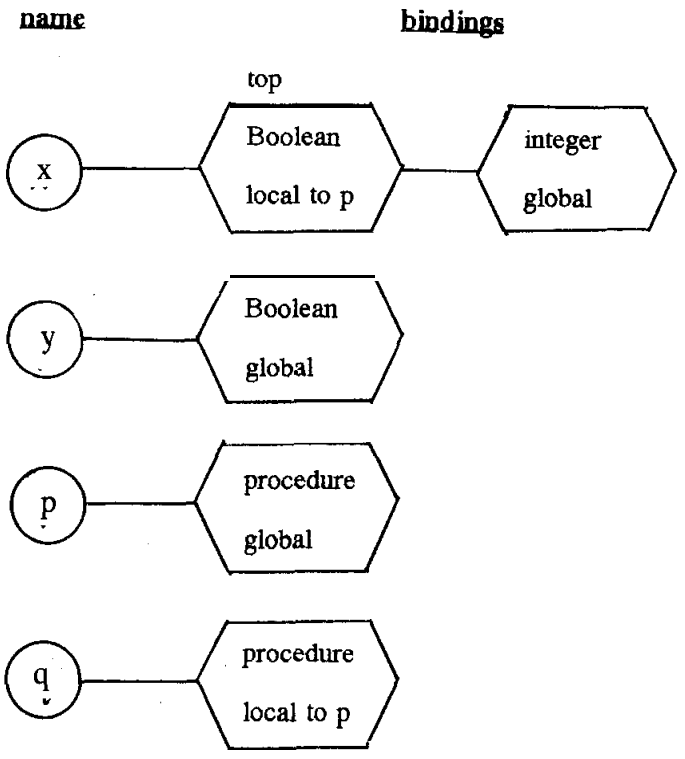
หลังจากประมวลผล การประกาศตัวแปร ของ p ตารางสัญลักษณ์ จะถูกแทนที่ดังนี้



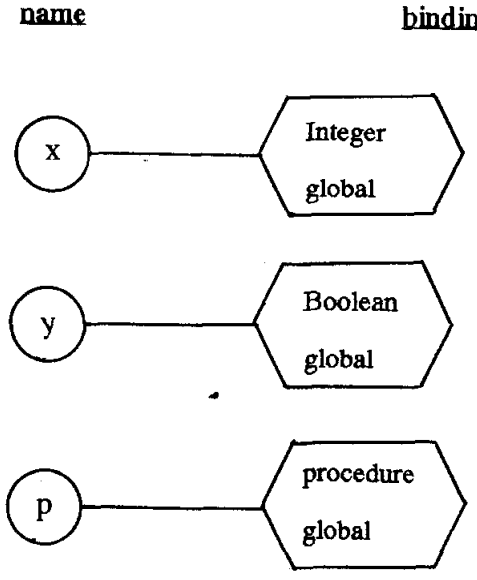
หลังจากประมวลผล การประกาศของ q ตารางสัญลักษณ์เปลี่ยนเป็นดังนี้



หลังจากปะมวลผล body ของ q การผูกโยง ซึ่งสร้างโดย การประกาศ ของ y local to q จะถูกลบออก จากกองซ้อน ชุดที่เกี่ยวข้องกับ ชื่อ y ดังนั้น ระหว่างการประมวลผล body ของ p ตารางสัญลักษณ์ เปลี่ยนเป็นดังนี้



สุดท้าย หลังจากประมวลผลบล็อก p การผูกโยงซึ่งสร้างโดย การประกาศของ x และ q local to p ถูกลบออก ดังนั้น ระหว่างการประมวลผล ของ โปรแกรมหลัก ตารางสัญลักษณ์ เปลี่ยนเป็นดังนี้



โปรดสังเกตว่า สิ่งนี้ รักษาสารสนเทศของสโคป (scope information) ที่เหมาะสม ได้แก่ scope holes สำหรับ การประกาศส่วนกลาง ของ x ภายใน p และการประกาศส่วนกลาง ของ y ภายใน q

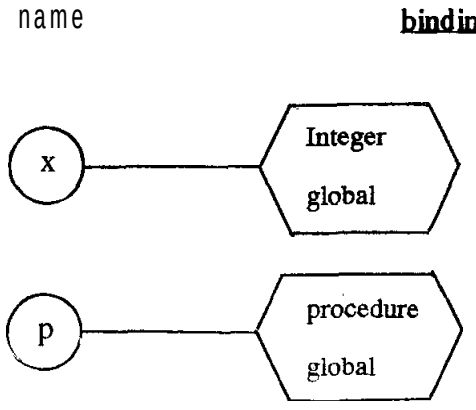
การแทนที่ของ ตารางสัญลักษณ์ข้างต้นนี้ สมมติว่า ตารางสัญลักษณ์ ประมวลผลการประกาศ แบบคงที่ นั่นคือ เกิดขึ้นก่อน การกระทำการ (prior to execution) สิ่งนี้คือ ตารางสัญลักษณ์ ถูกจัดการโดยตัวแปลโปรแกรม และการผูกโยงการประกาศ ทั้งหมดเป็นแบบคงที่

แต่ถ้าตารางสัญลักษณ์ ถูกจัดการในวิธีเดียวกันนี้ แต่เป็นแบบพลวัต นั่นคือ ระหว่างการกระทำการ (during execution) ดังนั้น การประกาศ ถูกประมวลผล ขณะที่ถูกพบ บนวิถีการกระทำการผ่านโปรแกรม ผลลัพธ์นี้อยู่ใน กฎสโคปที่แตกต่างกัน ซึ่งปกติเรียกว่า **dynamic scoping** และกฎ lexical scoping ชุดก่อนหน้าเรียกว่า static scoping

ตัวอย่าง ใน Pascal syntax ข้างล่างนี้ แสดงให้เห็นความแตกต่างระหว่าง scoping 2 ชนิดนี้

```
program ex;
var x : mteger;
procedure p;
begin
    writeln(x);
end;
procedure q;
var x : integer;
begin
    x := 2;
    p;
end;
begm (* main *)
    x := 1;
    q;
end.
```

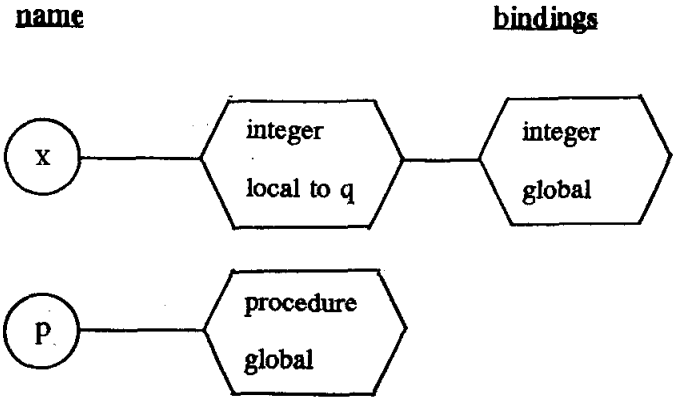
โดยวิธีก่อนหน้านี้ สำหรับ การแทนที่ตารางสัญลักษณ์ ภายใน p ตารางสัญลักษณ์ จะมีโครงสร้างดังนี้

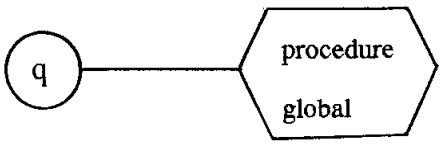


ดังนั้น การอ้างถึง x ในข้อความสั่ง writeln ของ p คือ x ส่วนกลาง และโปรแกรมจะพิมพ์ 1 สิ่งนี้ คือ lexical scoping ในทางตรงกันข้าม ระหว่างการกระทำการของ โปรแกรม p ถูกพบภายใน q วิถีการกระทำการเป็นดังนี้

main → call of q → call of p

ถ้าตารางสัญลักษณ์ ประมวลผลการประกาศ ขณะที่พบการประกาศ เหล่านี้ ในวิถีการกระทำนี้ ขั้นแรก การประกาศส่วนกลางของ main ถูกประมวลผล จากนั้น ประมวลผลการประกาศของ q และสุดท้าย ประมวลผลการประกาศของ p (ซึ่งในที่นี้ ไม่มีการประกาศใด) ดังนั้น ระหว่างการกระทำของ p โดยใช้ dynamic scoping ตารางสัญลักษณ์ จะเป็นดังนี้





ขณะนี้ การอ้างถึง x ภายในข้อความสั่ง writeln ของ p อ้างถึง local x ภายใน q เพราะว่า มันเป็นการประกาศครั้งหลังสุด ของ x ซึ่งจะถูกประมวลผลโดยตารางสัญลักษณ์ ดังนั้น โปรแกรม พิมพ์ 2 ไม่ใช่ พิมพ์ 1*

ภาษาโปรแกรมอื่นๆ ซึ่งใช้ dynamic scoping ได้แก่ APL, SNOBOL และภาษา LISP เวอร์ชันเก่า ภาษาสมัยใหม่เกือบทั้งหมด ใช้ lexical scoping จริงๆ แล้ว ขณะนี้ dynamic scoping พิจารณาแล้วว่ายังไม่น่าใช้ ด้วยเหตุผลหลายประการ คือ

ข้อแรก ภายใต้ dynamic scoping เมื่อชื่อหนึ่ง ถูกใช้ ใน นิพจน์ หรือ ข้อความสั่ง การประกาศซึ่งประยุกต์ใช้ ชื่อนั้น จะไม่ได้พบจากการอ่านโปรแกรม แต่ โปรแกรมต้องถูก กระทำการหรือ การกระทำการของมัน ตามรอย (traced) ด้วยมือ เพื่อหาการประกาศ ซึ่งจะประยุกต์ใช้

ข้อที่สอง dynamic scoping ชนกันกับ static typing ของ ตัวแปร ตัวอย่างเช่น จง พิจารณา โปรแกรมต่อไปนี้

```
program   barfoo;
var x : integer;

procedure          p;          .
begin
          x := ??;
end,
```

---

* เนื่องจาก ภาษา Pascal ใช้ lexical scope เราจึงไม่สามารถกระทำการ โปรแกรมนี้จริง ด้วยวิธีนี้ ตามข้อตกลง เราใช้ Pascal syntax เท่านั้น

```
procedure  q;
var  x  :  boolean;
begin
        p;
end;

begin  (* main  *)
        p;
        q;
end.
```

ระหว่างการกระทำการ เมื่อ p ถูกเรียกครั้งแรก (โดยโปรแกรมหลัก) การอ้างถึง x หมายถึง x ส่วนกลาง ซึ่งมีชนิดข้อมูล เป็น integer แต่มื่อ p ถูกเรียกในครั้งที่สอง (จากภายใน q) การอ้างถึง x หมายถึง x เฉพาะที่ ของ q ซึ่งมี ชนิดข้อมูล เป็น boolean ดังนั้น x อาจเป็นข้อมูลชนิด boolean หรือ ข้อมูลชนิด integer ได้ ภายใน p และ x ไม่สามารถถูกกำหนด ให้เป็นข้อมูลชนิด static ได้เพียงอย่างเดียว

จากการอภิปรายนี้ จึงดูเหมือนว่า เป็นสิ่งเป็นไปไม่ได้ที่จะรักษา lexical scope โดยใช้ตัวแปลคำสั่ง (interpreter) เพราะว่าจากบทนิยามของตัวแปลคำสั่ง ตารางสัญลักษณ์ จะเป็นแบบพลวัต สิ่งนี้ไม่เป็นปัญหา อย่างไรก็ตาม การรักษา lexical scoping แบบพลวัต ต้องการ โครงสร้างพิเศษ และ blockkeeping บางอย่าง ซึ่งรายละเอียด อยู่ในบทที่ 7 หัวข้อ environments และ procedure call

5.4 การจัดสรร ขนาด และสิ่งแวดล้อม

(Allocation, Extent, and Environment)

การพิจารณา รายละเอียด ของตารางสัญลักษณ์ เราจำเป็นต้องศึกษาสิ่งแวดล้อม ซึ่งเก็บรักษา การผูกโยงของชื่อ กับ ตำแหน่ง ทั้งนี้ขึ้นอยู่กับภาษา สิ่งแวดล้อม อาจสร้างขึ้นอย่างคงที่ (ณ เวลาบรรจุ) อย่างพลวัต (ณ เวลากระทำการ) หรือ เป็นการผสมกันทั้งสองแบบ

ภาษาซึ่งใช้สิ่งแวดล้อมแบบคงที่บริบูรณ์ ได้แก่ FORTRAN หมายถึง ตำแหน่งทั้งหมด

ถูกผูกโยงแบบคงที่

ภาษา ซึ่งใช้ สิ่งแวดล้อมแบบพลวัตบริบูรณ์ ได้แก่ LISP หมายถึง ทุกตำแหน่ง ถูกผูกโยง ระหว่าง การกระทำการ

ส่วนภาษา Pascal, C, Modula-2 และภาษารูปแบบ Algol อื่นๆ อยู่ตรงกลาง หมายถึง การจัดสรรหน่วยเก็บ บางอย่าง กระทำแบบคงที่ ในขณะที่ การจัดสรรอื่นๆ กระทำแบบพลวัต

ไม่ใช่ ชื่อทั้งหมด ใน โปรแกรม ซึ่ง ถูกผูกโยงให้กับตำแหน่ง ในภาษาแปลความ (compiled language) ชื่อของ constants และ data types อาจแทน purely compile-time quantities ซึ่งไม่มีอยู่ ณ เวลาบรรจุ หรือ เวลากระทำการ

ตัวอย่าง การประกาศ constant ของ Pascal

```
const   max = 10;
```

จะถูกใช้ โดย ตัวแปลโปรแกรม เพื่อแทนที่ การใช้ทั้งหมด ของ max ด้วยค่า 10 ชื่อ max จะไม่มีการจัดสรรหน่วยเก็บใดๆ และ รวมทั้งจะหายไปจากโปรแกรม เมื่อมัน กระทำการ

การประกาศ สามารถนำมาใช้ เพื่อสร้างสิ่งแวดล้อม เช่นเดียวกับ ตารางสัญลักษณ์ ในตัวแปลโปรแกรม (compiler) การประกาศ ใช้เพื่อแสดงว่า รหัสจัดสรรอะไรซึ่งตัวแปลโปรแกรม ใช้เพื่อ generate ขณะประมวลผลการประกาศ ในตัวแปลคำสั่ง (interprete) ตารางสัญลักษณ์ และสิ่งแวดล้อม เหมือนกัน ดังนั้น ลักษณะประจำ ซึ่งการผูกโยง โดยการประกาศ ในตัวแปลคำสั่ง รวม การผูกโยงตำแหน่ง ด้วย

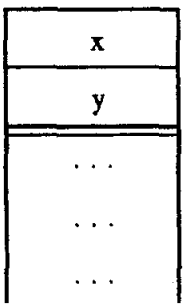
ขณะนี้เราจะ อภิปราย ความหลากหลายของการจัดสรร ใน ภาษาโครงสร้างแบบบล็อก (block-structured languages) และเล็กน้อยใน โครงสร้างของสิ่งแวดล้อม เราได้ตั้งข้อสังเกตมาแล้วว่า การจัดสรรอาจจะเป็น แบบคงที่ หรือแบบพลวัต โดยปกติ ตัวแปรส่วนกลาง ถูกจัดสรรแบบคงที่ เนื่องจากความหมายของมันคงที่ ตลอดทั้งโปรแกรม อย่างไรก็ตาม ตัวแปร เฉพาะที่ ให้กับบล็อก ซึ่งไม่ใช่บล็อกโปรแกรม ถูกจัดสรรแบบพลวัต เมื่อ การกระทำการ ถึงบล็อกนั้น ในหัวข้อสุดท้าย เราเห็นแล้วว่า ในภาษาเชิงโครงสร้างบล็อก ตารางสัญลักษณ์ ใช้ กลไก คล้ายกองซ้อน เพื่อเก็บ การผูกโยง ของการประกาศ ในทำนองเดียวกัน สิ่งแวดล้อม สำหรับภาษาเชิงโครงสร้างบล็อก ผูกโยง ตำแหน่ง ให้กับ ตัวแปรเฉพาะที่ ใน ลักษณะพื้นฐานกองซ้อน เพื่อให้เห็นว่า สิ่งนี้ เกิดขึ้นได้อย่างไร จงพิจารณาส่วนหนึ่ง ของ Algol60 ข้างล่างนี้ ซึ่งมีบล็อกซ้อนใน

```
A : begin
        integer  x;
        boolean  y;
        .
    B : begin
            real   x;
            integer  a;
            .
        end  B;
        .  .
    C : begin
            boolean  y;
            integer  b;
            .  .
        D : begin
                integer  x;
                real  y;
                .
            end  D;

        end  C;
        .
end  A;
```

ระหว่างการกระทำการ ของรหัสข้างต้นนี้ เมื่อเข้ามาในแต่ละบล็อก ตัวแปรซึ่งประกาศ ณ ตอนต้น ของแต่ละบล็อก จะถูกจัดสรร และเมื่อออกจากแต่ละบล็อก ตัวแปรตัวเดิม จะถูกคืนเนื้อที่ (deallocated) ถ้าเรามอง สิ่งแวดล้อมเป็น ลำดับเชิงเส้น ของ ตำแหน่ง หน่วยเก็บ (storage locations) ด้วยตำแหน่ง จัดสรร จาก บนสุดเรียงลำดับจากมากไปหาน้อย ดังนั้น สิ่งแวดล้อม

14-2

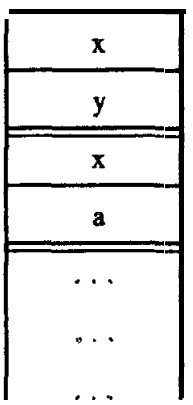
หลังจาก เข้าไปใน A จะเป็นดังนี้ (ไม่สนใจขนาดของ allocated variable แต่ละตัว)



Location bindings of A

(Unallocated space)

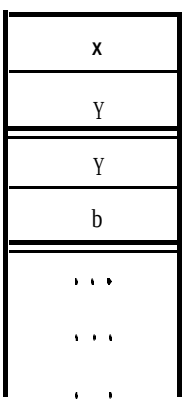
และสิ่งแวดล้อม หลังจาก เข้าไปใน B จะเป็นดังนี้



Location bindings of A

Location bindings of B

การออกจาก บล็อก B สิ่งแวดล้อม ส่งกลับ (returns) ไปยัง สิ่งแวดล้อม เหมือนขณะที่มีอยู่ หลังจาก เข้าไปใน A เมื่อเข้าไปในบล็อก C ตัวแปร ของ C ถูกจัดสรรเนื้อที่ และสิ่งแวดล้อม จะเป็นดังนี้

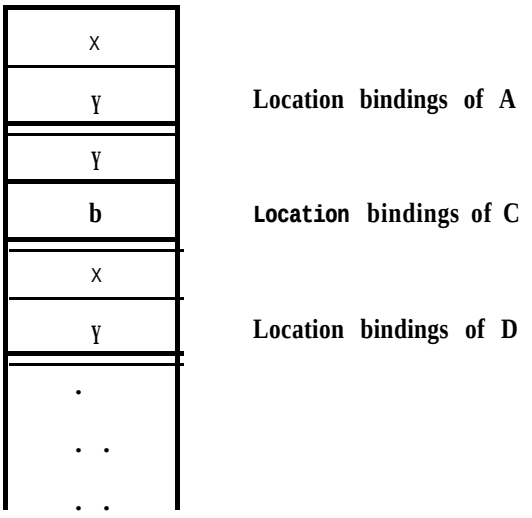


Location bindings of A

Location bindings of C

โปรดสังเกตว่า ตัวแปร y และ b ของบล็อก C ขณะนี้ ถูกจัดสรร ให้เป็นเนื้อที่เดียวกันกับเนื้อที่ก่อนหน้า ซึ่งเคยจัดสรร ให้กับ ตัวแปร x และ a ของบล็อก B สิ่งนี้ถูกต้องเพราะว่า เราอยู่นอก สโคป ของตัวแปรเหล่านี้ และ มันจะไม่ถูกอ้างถึงอีก

สุดท้าย การเข้ามายังบล็อก D สิ่งแวดล้อม จะเป็นดังนี้



การออกจาก แต่ละบล็อก ตำแหน่ง ซึ่ง ผูกโยงกับบล็อกนั้น จะถูกคืน (deallocated) อย่างสืบเนื่อง จนกระทั่ง ก่อน จะออกจาก บล็อก A อีกครั้งหนึ่งซึ่งเรามี recovered สิ่งแวดล้อมดังเดิม ของ A ในวิธีนี้ สิ่งแวดล้อม จึงคล้ายกับ กองซ้อน (วาดภาพ ของ สิ่งแวดล้อมเป็น "upside down" จาก กองซ้อนปกติ)

พฤติกรรม ของ สิ่งแวดล้อม ใน การจัดสรร และ การคืนเนื้อที่ สำหรับบล็อก begin-end ค่อนข้างง่าย ส่วนบล็อกโปรซีเดอร์ และบล็อกฟังก์ชัน จะซับซ้อนมากกว่า

ตัวอย่าง จงพิจารณา การประกาศโปรซีเดอร์ ของ Pascal ข้างล่างนี้

```
procedure  p;
var   x : integer;
      y : real;
begin
      . .
end; (* p *)
```

ระหว่างการกระทำการ เมื่อพบการประกาศนี้ บล็อกของ p จะยังไม่ถูกกระทำการ และ ตัวแปร เฉพาะที่ x และ y จะยังไม่ถูกจัดสรร ทั้งนี้ ตัวแปร x และตัวแปร y จะถูกจัดสรร ให้ก็ต่อเมื่อ p ถูกเรียกเท่านั้น ดังนั้น แต่ละครั้งที่ p ถูกเรียก ตัวแปรเฉพาะที่ตัวใหม่ จะถูกจัดสรร ดังนั้น การเรียก p แต่ละครั้ง ผลลัพธ์คือใน พื้นที่ของหน่วยความจำ จะถูกจัดสรรใน สิ่งแวดล้อม เราอ้างถึง การเรียก p แต่ละครั้งว่า เป็น **การใช้งานของ** p (activation of p) และพื้นที่ซึ่งสมนัยกัน ของ หน่วยความจำซึ่งถูกจัดสรร คือ **ระเบียนการใช้งาน** (activation record) การอธิบายโครงสร้าง ของ ระเบียนการใช้งาน ซึ่งสมบูรณ์มากกว่านี้ และสารสนเทศ ซึ่งจำเป็น เพื่อเก็บไว้ จะเลื่อนไปอภิปราย เรื่อง โปรซีเดอร์ ในบทที่ 7

จะเห็นชัดเจนจากตัวอย่างเหล่านี้ว่า ภาษาโครงสร้างแบบบล็อก ซึ่งมีสโคปแบบเชิงศัพท์ ชื่อเดียวกัน อาจ เกี่ยวข้องกับ ตำแหน่ง ต่างกันหลายแห่งได้ (แต่เฉพาะชื่อหน้าหนึ่งชื่อเท่านั้น ซึ่งจะถูกเข้าถึงในแต่ละครั้ง) ตัวอย่างเช่น ในสิ่งแวดล้อม ของ บล็อก Algol60 ในตัวอย่างข้างต้น ชื่อ x ผูกโยงกับ ตำแหน่ง สองแห่งที่แตกต่างกัน ระหว่างการ กระทำการ ของบล็อก D และชื่อ y ผูกโยง กับ ตำแหน่งที่แตกต่างกัน สาม แห่ง (แต่เฉพาะ x และ y ของ D เท่านั้น ซึ่งเข้าถึงได้ ณ เวลานั้น) เราต้องแยก ความแตกต่าง ระหว่างชื่อ ตำแหน่งที่จัดสรร และการประกาศ ซึ่งทำให้ สิ่งเหล่านี้ผูกโยงกัน

เราเรียกตำแหน่ง ซึ่งถูกจัดสรร ว่า **วัตถุ** นั่นคือ **วัตถุ** หมายถึง เนื้อที่ ของ หน่วยเก็บ ซึ่งถูกจัดสรร ใน สิ่งแวดล้อม ซึ่งเป็น ผลลัพธ์ ของการประมวลผล ของการประกาศ (That is, an **object** is an area of storage that is allocated in th environment as a result of the processing of a declaration.) จากบทนิยามนี้ ตัวแปร และ โปรซีเดอร์ ใน Pascal เป็น objects แต่ constants และ data types ไม่ใช่ objects (เพราะว่า การประกาศ ชนิด และค่าคงที่ ไม่มีผลลัพธ์ ในเรื่องการจัดสรรหน่วยเก็บ)

การมีอายุ หรือ extent ของ วัตถุ หมายถึง ช่วงเวลา ของ การจัดสรร ของมัน ใน สิ่งแวดล้อม

(The **lifetime** or **extent** of an object is the duration of its allocations in the environment.)

การมีอายุ ของ วัตถุ สามารถขยาย ภายใต้พื้นที่ของโปรแกรม ซึ่งมันอาจถูกเข้าถึงได้ ตัวอย่างเช่น ใน ตัวอย่าง Algol60 การประกาศของ integer x ในบล็อก A นิยาม วัตถุหนึ่งชิ้น ซึ่ง การมีอายุของมัน ขยายไปจนถึง บล็อก B ถึงแม้ว่า การประกาศ มี scope hole ใน B และ

วัตถุ ไม่สามารถเข้าถึงได้ จากภายใน B ในทำนองเดียวกัน มันเป็นไปได้ ที่จะย้อนลำดับ ของเหตุการณ์ที่เกิดขึ้น ดังนี้ :

วัตถุสามารถเข้าถึงได้ ภายใต้การมีอายุของมัน (An object can be accessible beyond its life time.)

ในภาษาโปรแกรม ซึ่งมี **ตัวชี้** (pointers) ให้ใช้ การขยายต่อไป ของ โครงสร้าง ของ สิ่งแวดล้อม เป็นเรื่องจำเป็น

ตัวชี้ หมายถึง วัตถุ ซึ่งค่าที่เก็บไว้ คือ ตำแหน่งของ วัตถุอีกสิ่งหนึ่ง

(A **pointer** is an object whose stored value is a reference to another object.)

ตัวอย่าง ในการประมวลผล การประกาศ ของ Pascal ข้างล่างนี้

```
var   x : ^integer;
```

จากสิ่งแวดล้อม ทำให้ มีการจัดสรร ตัวแปรชนิดตัวชี้ x ถึงแม้ว่า x จะถูกจัดสรร โดยการประกาศนี้ แต่ มันยังไม่ได้ชี้ ไปยัง วัตถุซึ่งจัดสรรแล้ว (allocated object) จริงๆ แล้ว x อาจมี undefined value ซึ่งคือตำแหน่งใดๆ ในหน่วยความจำ เพื่อทำให้ การเริ่มต้น ของตัวชี้ ซึ่งยังไม่ได้ชี้ไปยัง วัตถุจัดสรรแล้ว และยอมให้โปแกรม บอกว่า ตัวแปรชนิดตัวชี้ ชี้ไปยัง หน่วยความจำจัดสรร (allocated memory) หรือไม่ ภาษา Pascal มีตัวชี้ มีค่าเป็น nil ซึ่งจะถูกกำหนดค่า ให้กับ x เพื่อแสดงว่า มันยังไม่ได้ชี้ไปยัง **วัตถุจัดสรร** (allocated object) ใดๆ

ตัวอย่าง

```
x := nil;
```

ต่อมา x อาจถูกทดสอบ เพื่อดูว่า มันมีการจัดสรรหรือยัง

```
if  x<> nil  then  x^ := 2;
```

สำหรับ x ซึ่งชี้ไปยัง วัตถุจัดสรร เราต้องจัดสรรมัน โดยใช้ โปรซีเดอร์ **new** การเรียก

```
new(x);
```

หมายถึง จัดสรรตัวแปรจำนวนเต็ม ตัวใหม่ และเวลาเดียวกัน กำหนด ตำแหน่งของมัน ให้เป็นค่าของ x ตัวแปรจำนวนเต็มตัวใหม่นี้ เข้าถึงได้ โดยใช้ นิพจน์ x^ ทั้งนี้ ตัวแปร x เรียกว่า ถูก **dereferenced** โดยใช้ ตัวปฏิบัติการ "^" จากนั้น เราสามารถกำหนด ค่าจำนวนเต็ม ให้กับ x^ และอ้างถึง ค่าเหล่านี้ได้ เช่นเดียวกับ ที่เราใช้กับ ตัวแปรปกติ

ตัวอย่าง

```
x^ := 2;
```

```
writeln(x^);
```

การคืนการจัดสรร x^ ทำได้โดยเรียกโปรซีเดอร์ **dispose** ดังนี้

```
dispose(x);
```

สำหรับภาษา Modula-2 สถานการณ์ คล้ายกับ Pascal ยกเว้น โปรซีเดอร์ new และ dispose จะไม่มีให้ใช้อัตโนมัติ กล่าวคือ โปรซีเดอร์ ALLOCATE และ DEALLOCATE ถูกนำเข้ามาจาก **หน่วยเก็บของมอดูลคลัง** (library module storage) ดังนี้

```
MODULE Storageclient;
FROM Storage IMPORT ALLOCATE, DEALLOCATE;


END Storageclient.
```

ภายในมอดูล ซึ่งนำเข้า ALLOCATE และ DEALLOCATE นั้น วัตถุ ถูกชี้โดย ตัวแปรชนิดตัวชี้ ซึ่งสามารถ จัดสรรและคืนเนื้อที่จัดสรร ดังนี้

```
TYPE intptr = POINTER TO INTEGER;
VAR  x : intptr;
  . . .
ALLOCATE (x, SIZE(INTEGER));


DEALLOCATE (x, SIZE(INTEGER));
```

(ในการ implement ภาษา Modula-2 บางเวอร์ชัน อนุญาตให้ใช้ NEW และ DISPOSE อย่างเดียวกับที่ใช้ในภาษา Pascal แต่การตีความ นั้น เป็นอย่างเดียวกับ การเรียก ALLOCATE และ DEALLOCATE ดังนั้น โปรซีเดอร์เหล่านี้ ยังคงต้องนำเข้า)

สถานการณ์ ของ ภาษา C มีลักษณะคล้ายกัน คือการประกาศ

```
int * x;
```

สร้าง ตัวแปรชนิดตัวชี้จำนวนเต็มชื่อ x เริ่มต้นนั้น x เป็นตัวชี้ ซึ่ง ยังไม่ได้ชี้ ไปยัง เนื้อที่จัดสรร เขียนดังนี้

```
x = NULL:
```

(NULL มีความหมาย เหมือนกับ เลข 0 ซึ่งในภาษา C คือ integer 0 และตัวชี้ nil)

การจัดสรร วัตถุ และให้ x ชี้ที่ วัตถุตัวนี้ เราเรียก ฟังก์ชัน **malloc** (ย่อมาจาก **m**emory **alloc**ation) และกำหนด ค่าส่งคืนของมัน ให้กับ x ดังนี้

```
x = (int *) malloc (sizeof (int));
```

ฟังก์ชัน malloc ส่งคืนตำแหน่ง ซึ่ง มันจัดสรร และต้องกำหนดขนาดของ ข้อมูล ซึ่งมัน จะจัดสรรเนื้อที่ให้ด้วย สิ่งนี้กำหนดในรูปแบบ implementation-independent โดยใช้ฟังก์ชัน **sizeof** ซึ่งกำหนด ชนิดข้อมูล และขนาดของ ค่าส่งคืนของมัน (implemetation-dependent)

เมื่อกำหนดให้ วัตถุ ถูกชี้โดย x, เครื่องหมาย "*" ใช้สำหรับ dereference x (เหมือนกับ เครื่องหมาย "^" ในภาษา Pascal แต่ไว้คนละด้าน) เขียนดังนี้

```
*x = 2;
```

สุดท้าย ฟังก์ชัน **free** ซึ่งใช้ในภาษา C หมายถึง การ deallocate วัตถุซึ่งชี้โดย x เขียน ดังนี้

```
free(x);
```

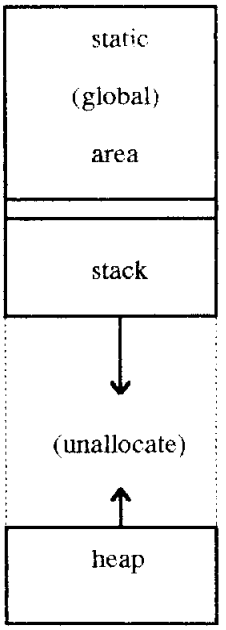
การที่จะให้มี การจัดสรร และการคืนการจัดสรร เนื้อที่ โดยใช้ new และ dispose (หรือ malloc และ free) นั้น สิ่งแวดล้อม ต้องมีเนื้อที่ ใน หน่วยความจำ ซึ่ง ตำแหน่ง สามารถถูกจัดสรร เพื่อโต้ตอบ กับการเรียกของ new และตำแหน่ง ซึ่งสามารถถูกส่งกลับ ในการโต้ตอบ กับการเรียก ของ dispose เนื้อที่เช่นนี้ เรียกว่า **ฮีป** (heap) (ถึงแม้ว่า จะ ไม่ได้ทำอะไรกับ โครงสร้างข้อมูล ฮีป)

การจัดสรรบนฮีป ปกติอ้างถึง **การจัดสรรแบบพลวัต** (dynamic allocation) แม้กระทั่ง การจัดสรร ของ ตัวแปรเฉพาะที่ เป็นแบบพลวัต เช่นกัน เพื่อแยกความแตกต่าง ของ การจัดสรร แบบพลวัต สองรูปแบบนี้ การจัดสรร ของ ตัวแปรเฉพาะที่ ตาม โครงร่าง แบบกองซ้อน ซึ่ง อธิบายในตอนแรกนั้น บางครั้งเรียกว่า **stack base** หรือ **automatic** เพราะว่า มันเกิดอย่าง อัตโนมัติ ภายใต้การควบคุม ของระบบเวลาดำเนินงาน (runtime system)

(เทอมที่เหมาะสมมากกว่า สำหรับ การจัดสรรตัวชี้ โดยใช้ new และ dispose คือ การจัด สรรด้วยมือ (manual allocation) เพราะว่า มันเกิดขึ้นภายใต้ การควบคุม ของ โปรแกรมเมอร์)

ในการ implement แบบปกติของสิ่งแวดล้อม กองซ้อน (สำหรับการจัดสรรแบบ อัตโนมัติ) และฮีป (สำหรับการจัดสรแบบพลวัต) ถูกเก็บที่ ปลายคนละด้าน ของ หน่วยความจำ ซึ่งให้ใช้ได้กับโปรแกรม และแต่ละด้านเพิ่มขึ้น เข้าหา อีกด้านหนึ่ง ขณะที่ ตำแหน่งใหม่ ถูกจัด

สรร โครงร่างของสิ่งแวดล้อม จึงเป็นภาพดังนี้



สรุป ในภาษาโครงสร้างแบบบล็อก ที่ใช้ตัวชี้ มีการจัดสรร ใน สิ่งแวดล้อม สาม ชนิดคือ

- static (สำหรับตัวแปรส่วนกลาง)
- automatic (สำหรับตัวแปรเฉพาะที่)
- dynamic (สำหรับตัวชี้)

การแบ่งประเภท อ้างถึง **storage class** ของตัวแปร บางภาษา เช่น C อนุญาตให้ การประกาศ ระบุ storage class ได้เช่นเดียวกับ แบบชนิดข้อมูล ปกติ สิ่งนี้ ใช้ใน C เพื่อเปลี่ยนแปลง การจัดสรร ของ ตัวแปรเฉพาะที่ ให้เป็น static

ตัวอย่าง

```
int f(void)
(static int x;
```

ขณะนี้ x ถูกจัดสรรเพียงครั้งเดียวเท่านั้น และมีความหมายเหมือนกัน (และค่า) ในทุกครั้งที่มีการเรียก f

ภาษา Algol68 มีการประกาศ storage class เช่นกัน รวมทั้ง การประกาศ กองซ้อน และ

ฮีป

เราจะยุติ (conclude) หัวข้อนี้ ด้วย ตัวอย่าง การใช้ สโคป และ extent เพื่อให้เห็นคุณสมบัติต่างๆ ของ ตัวแปรภายในโปรแกรม

ปัญหา จงเขียน ฟังก์ชัน ซึ่งส่งกลับ การนับ จำนวนครั้งที่ฟังก์ชันนี้ถูกเรียก (Write a function that returns a count of the number of times it is called.)

ผลเฉลย ภาษา Pascal ต่อไปนี้เป็นผลเฉลยภาษา Pascal (รวมทั้ง รหัสที่น่าสนใจบางอย่าง ในโปรแกรมหลัก เพื่อทดสอบการทำงาน)

```
program count;
const n = 10; (* or any other positive integer *)
var pcount : integer; (* the no of times p is called *)
          i : mteger;
function p : integer;
begin
    pcount := pcount + 1;
         p := pcount
end;
begin (* mam *)
    pcount := 0;
    for i := 1 to n do begin
        if p mod 3 <> 0 then writeln(p);
    end:
end.
```

ความยาก ของ ผลเฉลยนี้ คือไม่สามารถเขียน p ในลักษณะ self-contained มันขึ้นอยู่กับตัวแปรส่วนกลาง pcount ซึ่งต้องมีค่าเริ่มต้นในโปรแกรมหลัก จริงๆ แล้ว pcount ไม่สามารถทำให้เป็น local กับ p ดังนี้

```
function p : inetegr;
var pcount : inetegr; (* wrong! *)
begin
```

```
    pcount := pcount + 1:
        p := pcount;
end;
```

เพราะว่า มันทำให้ pcount เป็นอัตโนมัติ และค่าของมัน จะไม่เก็บ ผ่านการเรียก ของ p

ผลเฉลย ภาษา Modula-2

ตามที่เราตั้งข้อสังเกต ภาษา Modula-2 นั้น มอดูล จัดให้กับ สโคปที่จำกัด ฟังก์ชัน มอดูล คือ ขอบเขตของสโคป ในขณะที่เก็บคุณสมบัติ extent ของ บล็อกล้อมรอบมัน (A module functions as a scope boundary, while retaining the extent characteristics of its surrounding block.) ดังนั้น โปรแกรมนับ ซึ่งเขียนด้วย การออกแบบที่ดีกว่า ในภาษา Modula-2

```
MODULE Count;
FROM InOut IMPORT Writecard. Writeln;
        MODULE pcounter:
        EXPORT p;
        VXR pcount : CARDINAL;
        PROCEDURE p( ): CARDINAL:
        BEGIN
                INC(pcount);
                RETURN pcount;
        END p;
        BEGIN (* pcounter *)
                pcount := 0;
        END pcounter;
VAR i : CARDINAL;
BEGIN (* Count *)
        FOR i := 1 TO 10 DO
```

```
            IF p( ) MOD 3 <> 0 THEN
                Writecard(P( ), 1); Writeln;
            END; (* IF *)
        END; (* FOR *)
    END Count.
```

โปรซีเดอร์ p ถูกนำออกจาก มอดูล pcounter เพราะว่า กรณีอื่นๆ สโคปของมัน จะไม่ขยาย ไปยัง มอดูล Count ในทางตรงกันข้าม pcount จะไม่ถูกนำออก ดังนั้น มันจึง ไม่ถูกเปลี่ยนแปลงด้ว Count

ผลเฉลย ในภาษา C

ในภาษา C. storage class designator แบบ **static** อาจนำมาใช้ เพื่อให้ pcount ใส่ภายใน p ดังนั้นผลเฉลยที่ดีกว่า เขียนดังนี้

```
int  p(void)
(static int pcount = 0;
return (pcount += 1);}

void  main(void)
{ int i;
for (i = 1; i <= 10; i++)
    [if (p( ) %3 != 0) printf("%d\n", p( ));}
```

ผลเฉลยข้างต้นนี้ pcount ไม่สามารถ ถูกอ้างถึง นอก p เพราะว่า มัน local กับ p อย่างไรก็ตาม pcount ถูกจัดสรรแบบคงที่ และยังคงเก็บค่าของมัน ผ่านการเรียก p (เริ่มต้น p = 0 กระทำเพียงครั้งเดียว สำหรับตัวแปร แบบคงที่)

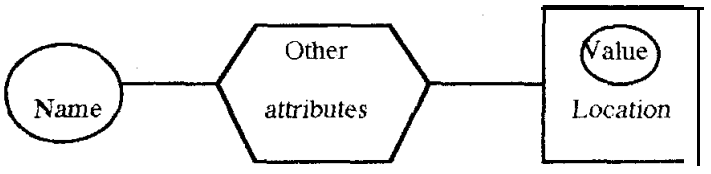
6.6 **ตัวแปรและตัวคงที่** (Variables and Constants)

5.5. **ตัวแปร** (Variables)

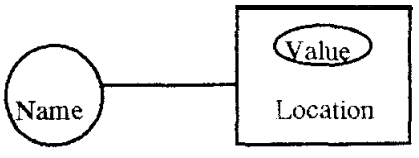
ตัวแปร หมายถึง วัตถุ ซึ่ง ค่าของมันที่เก็บไว้ เปลี่ยนแปลงได้ ระหว่างการกระทำการ (A **variable** is an object whose stored value can change during execution.)

ตัวแปร จะถูกกำหนดรายละเอียดครบถ้วนด้วย **ลักษณะประจำ** (attributes) ของมัน ได้แก่ ชื่อ (name) ตำแหน่ง (location) ค่า (value) และลักษณะประจำอื่นๆ ของมัน เช่น แบบ**ชนิดข้อมูล** (datatype) และ ขนาด (size)

การแทนที่ โครงร่างของตัวแปร วาดภาพดังนี้ (A schematic representation of a variable can be drawn as follows:)



ภาพข้างบนนี้ มี ชื่อ ตำแหน่ง และค่าของตัวแปร ซึ่งเป็นลักษณะประจำ ที่สำคัญของมัน บ่อยครั้งที่เรา ต้องการเน้นเฉพาะสิ่งเหล่านี้ ดังนั้น ภาพของตัวแปร หนึ่งตัว จะเป็นดังนี้



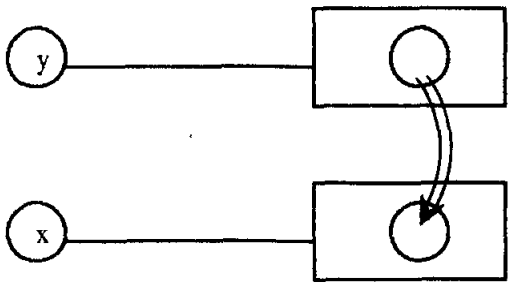
เราเรียกสิ่งนี้ว่า **แผนภาพ กล่องและวงกลม** (box-and-circle diagram) เส้นตรงที่ลากเชื่อม ระหว่าง ชื่อ และกล่องตำแหน่ง แทน การผูกโยง ของ ชื่อ กับ ตำแหน่ง โดยสิ่งแวดล้อม และวง-กลม ภายในกล่อง แทนค่า ซึ่งผูกโยง ด้วย หน่วยความจำ นั่นคือ ค่าซึ่งเก็บ อยู่ ณ ตำแหน่งนั้น

ตัวแปร เปลี่ยนแปลงค่าของมัน ด้วย ข้อความสั่งกำหนดค่า ตัวอย่างเช่น x := e เมื่อ x เป็นชื่อตัวแปร และ e เป็น นิพจน์

ความหมาย (semantics) ของ ข้อความสั่งนี้คือ e ถูก ประเมินผล ให้เป็น ค่าหนึ่งค่า ซึ่ง จากนั้น จะถูกทำสำเนา ไปยัง ตำแหน่งของ x ถ้า e เป็นชื่อตัวแปร (variable name) สมมติ ให้เป็น y ดังนั้น การกำหนดค่า

x := y

จะเขียนภาพดังนี้ (ลูกศรสองเส้น หมายถึง การทำสำเนา)



เนื่องจาก ตัวแปร มีทั้ง ตำแหน่ง และค่าซึ่งเก็บในตำแหน่งนั้น จึงสำคัญ ที่จะแบ่งสองสิ่งนี้ ให้ชัดเจน อย่างไรก็ตาม ความแตกต่างนี้ จะมองไม่เห็น ใน ข้อความสั่งกำหนดค่า กล่าวคือ y อยู่ทางด้านขวามือ หมายถึง ค่าของ y ในขณะที่ x อยู่ทางด้านซ้ายมือ หมายถึงตำแหน่งของ x ด้วยเหตุผลนี้ ค่าซึ่งเก็บ ในตำแหน่งของตัวแปร บางครั้ง เรียกว่า r-value (สำหรับ ค่าทางขวามือ) ในขณะที่ ตำแหน่งของตัวแปร เรียกว่า l-value (สำหรับ ค่าทางซ้ายมือ)

บางภาษา เช่น Algol68 และ BLISS ทำ ข้อแตกต่าง ชัดเจนมากกว่า ระหว่าง r-values, และ l-values, ในภาษา Algol68 นั้น ปกติ ชื่อตัวแปร หมายถึง ตำแหน่งของมัน ตั้งแต่แรก หรือ l-value ส่วนการเข้าถึง r-value ตัวแปร ต้องถูก **dereferenced**

ตัวอย่าง การประกาศ

```
int x;
```

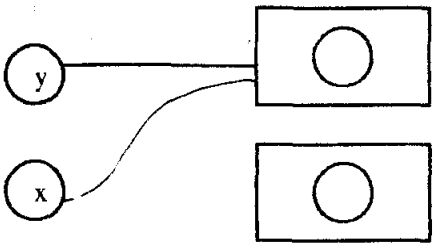
หมายถึง ประกาศ x ให้เป็น การอ้างถึง **(reference) integer** (ภาษา Algol68 เรียกว่า **ref int** mode) การกำหนด ค่าของ x ให้กับ ตัวแปร y ชนิด **integer** อีกตัวหนึ่ง จะต้อง **dereferenced** x ดังนี้ :

```
y := (int) x;
```

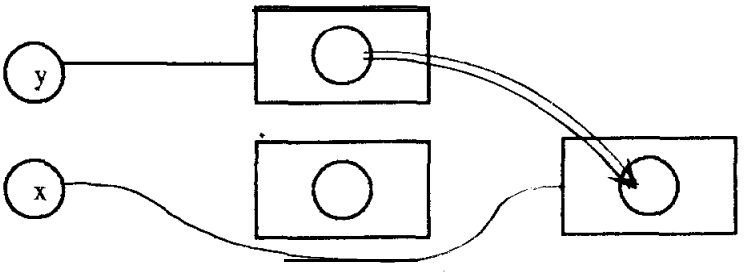
(นิพจน์ ชนิดที่มีวงเล็บกำกับ หมายถึง **cast**) มีกฎ ซึ่งยอมให้ **automatic dereferencing** เช่น ในภาษา Pascal

ในบางภาษา ความหมายที่แตกต่างกัน ถูกกำหนดให้กับ การกำหนดค่า ดังนี้ :

ตำแหน่งถูกทำสำเนา แทนที่ จะเป็นการทำสำเนาค่า (:locations are copied instead of values.) ในกรณีนี้ x := y ผลลัพธ์คือ การผูกโยง ตำแหน่ง ของ y ให้กับ x แทนที่จะเป็น การผูกโยงค่าของมัน



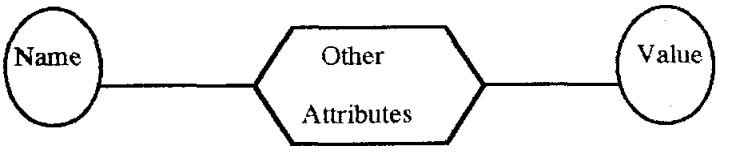
สิ่งนี้ คือ **assignment by sharing** ทางเลือกอีกอย่างหนึ่งคือ จัดสรร ตำแหน่งใหม่ แล้ว ทำสำเนาค่าของ y และผูกโยง x ให้กับ ตำแหน่งใหม่ ดังนี้ :



ทั้งสองกรณี การตีความหมาย ของ การกำหนดค่า บางครั้ง หมายถึง **pointer semantics** เพื่อแยกความแตกต่างมัน จากความหมายปกติ ภาษาโปรแกรม ซึ่งใช้ pointer semantics สำหรับ การกำหนดค่า ได้แก่ SNOBOL ในบางกรณี ภาษา LISP ใช้ pointer semantics ด้วยเช่นกัน

5.5.2 **ตัวคงที่** (Constants)

ตัวคงที่ หมายถึง เอนทิตี ของภาษา ซึ่งมีค่าคงที่ สำหรับ ช่วงเวลาของ โปรแกรม (A **constant** is a language entity that has a fixed value for the duration of the program.) ตัวคงที่ เหมือนกับตัวแปร ยกเว้น ตัวคงที่ ไม่มีตำแหน่งชัดแจ้ง มีแต่ค่าเท่านั้น



บางครั้ง เราพูดว่า ตัวคงที่ มี **value semantics** แทนที่จะเป็น storage semantics เหมือน ในตัวแปร สิ่งนี้ ไม่ได้หมายความว่า ตัวคงที่ ไม่ได้ถูกเก็บในหน่วยความจำ สิ่งที่เป็นไปได้ คือ ตัวคงที่ มี ค่าหนึ่งค่า ซึ่งทราบค่าเฉพาะ ณ เวลากระทำการเท่านั้น (It is possible for a constant to have a value that is known only at execution time.)

ในกรณีนี้ ค่าของมัน ต้องเก็บ ใน หน่วยความจำ แต่ที่ไม่เหมือนกับ ตัวแปร คือ เมื่อค่านี้ถูกคำนวณ มันจะไม่เปลี่ยนแปลง และตำแหน่งของตัวคงที่ จะไม่ถูก อ้างถึงชัดแจ้ง ด้วยโปรแกรม

แนวคิดนี้ ของ ตัวคงที่ คือ การเป็น **สัญลักษณ์** (symbolic) นั่นคือ ตัวคงที่ จำเป็น จะต้องมี ชื่อ ให้กับ ค่าหนึ่งค่า บางครั้ง การแทนที่ ของค่า เช่น ลำดับ ของเลขโคด 42 หรือ การแทนที่ ของ อักขระหนึ่งตัว เช่น "a" เรียกว่า ตัวคงที่ เพื่อ แยกสิ่งนี้ ให้มีความแตกต่างกัน ในการประกาศตัวคงที่ บางครั้งเราเรียกการแทนที่ของค่าว่า **สัญพจน์ (literals)** ตามที่ได้ตั้งข้อสังเกตมาแล้ว ตัวคงที่ อาจจะเป็นแบบ คงที่ หรือแบบพลวัต ก็ได้ ตัวคงที่แบบคงที่ หมายถึง ค่าของมันถูกคำนวณ ก่อนการกระทำการ ในขณะที่ ตัวคงที่แบบพลวัต ค่าของมันถูกคำนวณเฉพาะ ระหว่างการกระทำการ เท่านั้น

(A **static constant** is one whose value can be computed prior to execution, while a dynamic constant has a value that can be computed only during execution.)

ภาษา Pascal มีเฉพาะ constants แบบคงที่เท่านั้น โดยมีข้อจำกัดว่า เฉพาะ literal ซึ่งเป็นค่าของตัวคงที่ เท่านั้นที่ใช้ได้ ในการประกาศตัวคงที่ ดังนั้น ในการประกาศ

```
const size = 42;
      max = size - 1;  (* illegal Pascal *)
```

การประกาศของ size ถูกต้อง แต่การประกาศของ max ไม่ถูกต้อง เพราะว่าค่า ซึ่งกำหนดให้กับ max ไม่ใช่ literal แต่เป็น นิพจน์ โปรดสังเกตว่า max ยังคงเป็น static constant

ตัวแปลโปรแกรมภาษา Pascal มอง ตัวคงที่ เป็น **ชื่อสัญลักษณ์** (symbolic names) สำหรับ ค่าต่างๆ ซึ่ง กำหนด โดยตรง ในการประกาศตัวคงที่ และการเกิดทั้งหมด ของ ชื่อสัญลักษณ์ ใน โปรแกรม จะถูกแทนที่ ทันที โดยค่าของมัน ด้วยเหตุผลนี้ ตัวคงที่ของ Pascal บางครั้ง เรียกว่า **manifest constants**

ภาษา Modula-2 ไม่มีขีดจำกัด บน ตัวคงที่ ดังนั้น นิพจน์ อาจปรากฏในการประกาศตัวคงที่ได้

ตัวอย่าง การประกาศตัวคงที่ ทั้งหมด ข้างล่างนี้ ถูกต้องในภาษา Modula-2

```
CONST twoplusthree = 2 + 3;
      size = 42;
      max = size - 1;
      a   = twoplusthree * size DIV 5;
```

```
e    =  2.718336:
x    =  1.0/e;
```

ตัวคงที่ใน Modula-2 ยังคงเป็น static อย่างไรก็ตาม การประกาศ

```
CONST  pi = 4.0 * arctan (1.0);
```

ผิดใน Modula-2 เพราะว่า รหัส สำหรับฟังก์ชัน arctan จะไม่ทราบค่า ณ เวลาแปลโปรแกรม (มันถูกนำเข้า จาก มอดูลคลัง และถูกโยง หลังจากการแปล) จริงๆแล้ว ภาษา Modula-2 มีการจำกัด constant expressions เพื่อไม่ให้มันเป็น function calls หรือ predefined functions ดังนั้น การประกาศ ข้างล่างนี้ ผิด เช่นกัน ในภาษา Modula-2

```
CONST Null = CHR(0);
```

เหตุผลคือ CHR ไม่ได้ สำรองไว้ และ อาจถูกแทนที่ด้วย an unknown user-defined function

ภาษา Algol68 และ Ada อนุญาต ให้ใช้ dynamic constants ได้

ตัวอย่าง

```
pi : constant FLOAT := 4.0 * arctan (1.0);
```

เป็น ตัวคงที่ซึ่งถูกต้องของภาษา Ada ดังนั้น การประกาศ ตัวคงที่เฉพาะที่ ข้างล่างนี้ ถูกต้องเช่นกัน

```
procedure Swap(x, y : in out INTEGER) is
    temp : constant INTEGER := x;
begin
    x := y
    y := temp
end Swap;
```

5.6 สมนาม การนำมาใช้ไม่ได้ และขยะ

(Aliases, Dangling references, and Garbage)

หัวข้อนี้ จะอธิบาย ปัญหา หลายๆ อย่าง ซึ่ง เกิดขึ้น จากการ ตั้งชื่อ (naming) และข้อตกลงการจัดสรรเนื้อที่ ของ ภาษาโปรแกรมต่างๆ โดยเฉพาะ ภาษาโครงสร้างแบบบล็อก เช่น Pascal, Modula-2, Ada และ C

ผลเฉลย การออกแบบภาษา อาจพบปัญหาเหล่านี้ จำนวนมาก ซึ่งตรงกันข้าม กับ ผลเฉลย ของ โปรแกรมเมอร์ ซึ่งหลีกเลี่ยง สถานการณ์ เชิงปัญหา และมีการอภิปราย ถึงสิ่งเหล่านี้ ไม่มาก นัก

5.6.1 **สมนาม** (Aliases)

สมนามเกิดขึ้น เมื่อ สิ่งของหนึ่งอย่าง ผูกโยงให้กับ ชื่อสองชื่อที่แตกต่างกัน ณ เวลา เดียวกัน

(An alias occurs when the same object is bound to two different names at the same time.)

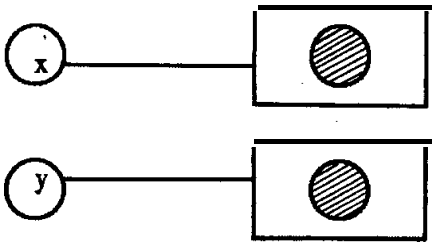
สมนามเกิดขึ้นได้หลายวิธี วิธีหนึ่ง คือระหว่างการเรียกโปรซีเดอร์ ซึ่งจะศึกษาในบทที่ 7 อีกวิธีหนึ่งคือ ผ่านทาง การใช้ตัวแปรชนิดตัวชี้ (the use of pointer variables)

ตัวอย่าง ภาษา Pascal

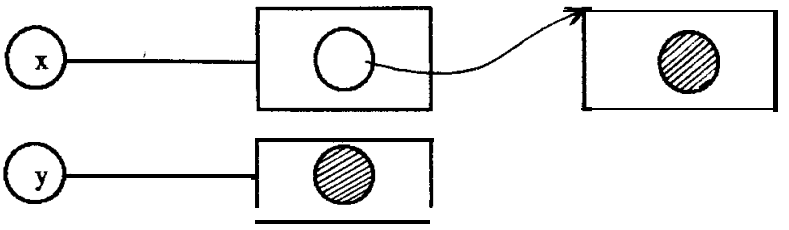
```
type intptr = *integer;
var x, y : intptr;
begin
    new(x);
    x^ := 1;
    y := x; (* x^ and y^ now aliases *)
    y^ := 2;
    writeln(x^);
end;
```

หลังจาก การกำหนดค่า ของ x ให้กับ y ขณะนั้น ทั้ง y^ และ x^ หมายถึง ตัวแปรตัว เดียวกัน และรหัสข้างต้น เอ้าพุทคือ พิมพ์ 2 เราสามารถดูสิ่งนี้ได้ชัดเจนขึ้น ถ้า บันทึกผล ของ รหัสข้างต้น ด้วย แผนภาพกล่อง-และ-วงกลม ดังนี้

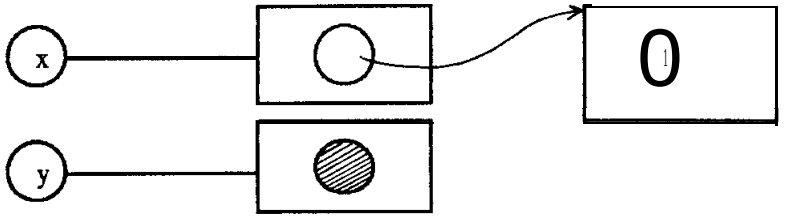
หลังจากการประกาศ x และ y ทั้งคู่ มีการจัดสรรเนื้อที่ให้ ในสิ่งแวดล้อม แต่ค่า ของทั้งคู่ ยัง undefined แสดงด้วยแผนภาพข้างล่างนี้ การแรเงา หมายถึง ค่า



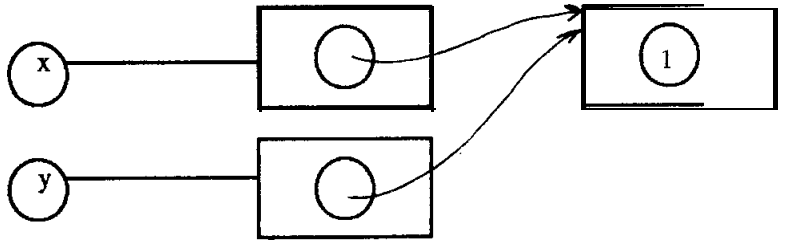
หลังจากเรียก new(x) จะมีการจัดสรรเนื้อที่ ให้ x^ และ x จะมีการกำหนดค่า ให้เท่ากับตำแหน่ง ของ x^ แต่ตัว x^ ยังคง undefined แสดงด้วยภาพดังนี้



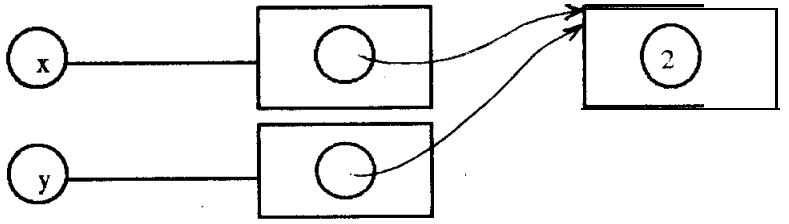
หลังจาก การกำหนดค่า x^ := 1 สถานการณ์ เป็นดังนี้



การกำหนดค่า y := x ขณะนี้ ทำสำเนา ค่าของ x ให้กับ y และทำให้ y^ และ x^ เป็นสมนาม ซึ่งกันและกัน (โปรดสังเกตว่า x และ y **ไม่ใช่** สมนาม ซึ่งกันและกัน)



สุดท้าย การกำหนด่า y^ := 2 ผลลัพธ์ คือแผนภาพข้างล่างนี้



สมนาม นำเสนอ ปัญหา ซึ่งทำให้เกิด **ผลกระทบข้างเคียง** (side effects) ที่เป็นอันตรายมาก สำหรับเป้าหมายของเรา เรานิยาม ผลกระทบข้างเคียง ของข้อความสั่ง หมายถึง การ

เปลี่ยนแปลงใดๆ ในค่าของตัวแปร ซึ่งอยู่ภายใต้การกระทำการของข้อความสั่ง (We define a side effect of a statement to be any change in the value of a variable that persists beyond the execution.) จากบทนิยามนี้ ผลกระทบข้างเคียง จึงไม่เป็นอันตรายทั้งหมด เพราะว่า การกำหนดค่า คือความตั้งใจอย่างชัดแจ้ง ให้เกิดสิ่งหนึ่ง อย่างไรก็ตาม ผลกระทบข้างเคียง ซึ่งเปลี่ยนแปลงตัวแปรต่างๆ ซึ่ง ชื่อของมัน ไม่ได้ปรากฏ โดยตรง ใน ข้อความสั่งนั้น เป็นอันตรายมาก ในแง่ที่ว่า ผลกระทบข้างเคียง ไม่สามารถบอกได้ จากรหัสที่เขียน ในตัวอย่างข้างต้น การกำหนดค่า y^ = 2 เปลี่ยนแปลง x^ ถึงแม้ว่า การเปลี่ยนแปลงนี้ จะไม่ปรากฏ ในข้อความสั่ง ซึ่งเป็นเหตุให้เกิดขึ้น

สมนาม อันเนื่องมาจาก การกำหนดค่าของตัวชี้ ควบคุมยาก และเป็นเหตุผลหนึ่ง ในหลายๆ เหตุผล ซึ่งทำให้การเขียนโปรแกรม ด้วยตัวชี้ ยากมาก ภาษาโปรแกรม ภาษาหนึ่ง ซึ่งพยายามที่จะจำกัด การสมนาม ไม่ใช่เพียงแต่ตัวชี้ แต่ทั้งภาษา ได้แก่ ภาษา Euclid

ภาษา FORTRAN มีกลไกชัดแจ้ง สำหรับ การสมนาม โดยใช้ ข้อความสั่ง EQUIVALENCE

ตัวอย่าง

```
EQUIVALENCE  X, Y
```

ทำให้ X และ Y เป็นสมนามซึ่งกันและกัน ดังนั้น การกำหนดค่า ให้กับ X จึงเป็นเหตุให้มีการกำหนดค่าโดยนัย ให้กับ Y ด้วย และสิ่งนี้เป็นจริงในทางย้อนกลับ ในสมัยแรกๆ ของการเขียนโปรแกรม ข้อความสั่ EQUIVALENCE นำมาใช้เพื่อ ลด (reduce) ปริมาณของหน่วยความจำ ซึ่งจำเป็น ต้องใช้ สำหรับโปรแกรม ขนาดใหญ่ โดยการ sharing ตำแหน่งหน่วยความจำระหว่างตัวแปรต่างๆ ซึ่งไม่ได้ใช้ ณ เวลาเดียวกัน ทุกวันนี้ การใช้ ข้อความสั่งนี้ ถือว่าเป็นการออกแบบโปรแกรมที่แย่ (poor design) ข้อความสั่ง COMMON ในภาษา FORTRAN อาจทำให้เกิดการสมนาม ได้เช่นกัน (ดู แบบฝึกหัดข้อ 3)

5.6.2 การนำมาไม่ได้ (Dangling References)

การนำมาไม่ได้ เป็นปัญหาข้อที่สอง ซึ่ง อาจเกิดขึ้นได้ จากการใช้ตัวชี้

การนำมาไม่ได้ หมายถึง ตำแหน่งนั้น มีการ คืนเนื้อที่ไปแล้ว จากสิ่งแวดล้อม แต่ยังคงมีการเข้าถึงโดยโปรแกรม

(A dangling reference is a location that has been deallocated from the environment, but that can still be accessed by a program.)

อีกวิธีหนึ่ง ของการกล่าวถึงสิ่งนี้คือ การนำมาใช้ไม่ได้ เกิดขึ้น ถ้าวัตถุสิ่งหนึ่ง ถูกเข้าถึง ภายหลังจาก การสิ้นสุดอายุของมัน ในสิ่งแวดล้อม

(An other way of stating this is that a dangling reference occurs if an object can be accessed **beyond** its lifetime in the environment.)

ตัวอย่างง่ายๆ ของ การนำมาใช้ไม่ได้คือ ตัวชี้หนึ่งตัว ซึ่งชี้ไปยัง วัตถุซึ่งคืนเนื้อที่ไปแล้ว

(A simple example of a dangling reference is a pointer that points to a deallocated object.)

ในอย่าง Pascal การใช้ โปรซีเดอร์ dispose ทำให้เกิด การนำมาใช้ไม่ได้ ดังนี้

```
type   intptr = *integer;
var x, y : intptr;
   .
new(x);
   . . .
x^ := 2;

y := x;          (* y^ and x^ now aliases *)
dispose(x);      (* y^ now a dangling reference *)

writeln(y^);     (* illegal! *)
```

ในภาษา C การนำมาใช้ไม่ได้ อาจเป็นไปได้เช่นกัน เนื่องจากการคืนการจัดสรรอัตโนมัติ ของ ตัวแปรเฉพาะที่ เมื่อออกจาก บล็อกของการประกาศเฉพาะที่ ทั้งนี้เพราะว่า ภาษา C มี แอดเดรส ของ ตัวปฏิบัติการ "&" ซึ่ง อนุญาตให้ ตำแหน่งของตัวแปรใดๆ ก็ตาม ถูกกำหนดค่าให้กับ ตัวแปรชนิดตัวชี้ได้

ตัวอย่าง ภาษา C

```
{int  *x;
   {int   y;
      y = 2;
```

```
x = &y;}
/* *x is now a dangling reference */
```

เมื่อเราออกจาก บล็อก ซึ่ง y ถูกประกาศ อยู่ภายในตัวแปร x ประกอบด้วยตำแหน่งของ y และตัวแปร *x เป็นสมนามกับ y แต่ในสิ่งแวดล้อม ขึ้นอยู่กับ กองซ้อนมาตรฐาน ซึ่งอธิบายมาแล้วในหัวข้อ 5.4 เมื่อออกจากบล็อก y จะถูกคืนการจัดสรรเนื้อที่

ตัวอย่าง ภาษา C

```
int * dangle (void)
{int x;
 return &x; }
```

เมื่อ ฟังก์ชัน dangle ถูกเรียก มันจะส่งคืน ตำแหน่งของ ตัวแปรอัตโนมัติเฉพาะที่ x ของมัน ซึ่งเพิ่งถูกคืนการจัดสรร ดังนั้น หลังจากการกำหนดค่าใดๆ เช่น

```
y = dangle( )
```

ตัวแปร *y จะเป็น dangling reference

ภาษา Pascal ไม่มี dangling reference ชนิดนี้ เพราะว่า ภาษานี้ไม่มี ฟังก์ชัน ซึ่งมีความหมาย เหมือนกับ ฟังก์ชัน "&" ของภาษา C

ภาษา Modula-2 มีฟังก์ชันที่คล้ายกัน เรียกว่า ADR อย่างไรก็ตาม สิ่งนี้ และสิ่งอำนวยความสะดวก ระดับต่ำอื่นๆ จะนำมาใช้ในโปรแกรมได้ ก็ต่อเมื่อ มันถูกนำเข้าอย่างชัดแจ้ง จากมอดูล SYSTEM และมอดูล ซึ่งนำเข้าจาก SYSTEM จะตั้งข้อสังเกตได้ว่า อาจกำลังจะ inherently unsafe

5.6.3 ขยะ (Garbage)

วิธีที่ง่ายวิธีหนึ่งของการ ขจัด ปัญหา dangling reference คือ ไม่ต้อง คืนการจัดสรรใดๆ ทั้งสิ้น จากสิ่งแวดล้อม แต่สิ่งนี้ ทำให้เกิดปัญหาที่สาม คือ **ขยะ** ซึ่งจะได้อภิปรายในหัวข้อนี้

ขยะ หมายถึง หน่วยความจำ ซึ่งมีการจัดสรรให้ ในสิ่งแวดล้อม แต่หน่วยความจำนี้ โปรแกรม เข้าถึงไม่ได้

(Garbage is memory that has been allocated in the environment but that has become inaccessible to the program.)

ในภาษา Pascal วิธีหนึ่งของการเกิดขยะ คือไม่เรียก dispose ก่อน การกำหนดค่าใหม่ให้กับ pointer variable

ตัวอย่าง

```
var   x : ^integer
   . . .
new(x);
x := nil;
```

ตอนจบของรหัสข้างต้นนี้ ตำแหน่งซึ่ง จัดสรรให้กับ x^ โดยเรียกจาก new(x) ขณะนี้คือ ขยะ เพราะว่า x เป็นตัวชี้ nil และไม่มีวิธีใด ที่จะเข้าถึง วัตถุ ซึ่งจัดสรรก่อนหน้านี้

อีกสถานการณ์หนึ่ง ซึ่งคล้ายกัน เกิดขึ้นเมื่อ การกระทำการ ทิ้ง (leaves) พื้นที่ ของ โปรแกรม ซึ่ง ตัว x ถูกจัดสรรแล้ว ดังนี้

ตัวอย่าง ภาษา Pascal

```
procedure  p;
var  x : ^integer;
begin
    new(x);
    x^ := 2;
end;
```

เมื่อออกจาก โปรซีเดอร์ p ตัวแปร x ถูกคืนการจัดสรร และ x^ ไม่สามารถเข้าถึงได้อีกต่อไป โดยโปรแกรม สถานการณ์ที่คล้ายกันนี้ เกิดขึ้น ใน บล็อก ชนิดอื่นๆ ตัวอย่างเช่น ส่วนหนึ่งของภาษา C ข้างล่างนี้

```
{ . . .
    {int * x;
        x = (int *) malloc(size of (int));
        . . .}
        /*  *x no longer accessible here */
}
```

ขยะ เป็นปัญหาในการกระทำการของโปรแกรม เพราะว่ามันทำให้ หน่วยความจำสูญ

เปล่า

(Garbage is a problem in program execution because it is wasted memory.)

อย่างไรก็ตาม การโต้แย้ง อาจเกิดขึ้นที่ว่า โปรแกรม ซึ่ง ทำให้เกิด (produce) ขยะ อาจจะเป็นปัญหาที่สำคัญ (seriously) น้อยกว่า โปรแกรม ซึ่ง ประกอบด้วย dangling references

โปรแกรม ซึ่ง ทำให้เกิด ขยะ อาจ วิ่ง (run) ไม่ได้ เพราะว่า หน่วยความจำ ไม่พอ แต่มันถูกต้องอย่างภายใน นั่นคือ ถ้าหน่วยความจำ มีมากพอ ให้ใช้ มันจะให้ผลลัพธ์ ถูกต้อง (หรืออย่างน้อยที่สุด ไม่ผิด เพราะว่า ล้มเหลว ที่จะคืนเนื้อที่หน่วยความจำ ซึ่งเข้าถึงไม่ได้)

ในทางตรงกันข้าม โปรแกรม ซึ่ง เข้าถึง dangling references อาจวิ่งได้ (run) แต่ให้ผลลัพธ์ผิด หรืออาจ แย่ง (corrupt) หน่วยความจำ จาก โปรแกรมอื่นๆ หรืออาจเกิด ข้อผิดพลาดเวลาดำเนินงาน (runtime errors) ซึ่งยากที่จะหาตำแหน่งที่เกิด

ด้วยเหตุผลนี้ จึงเป็นประโยชน์ ที่จะลบทิ้ง ความจำเป็น ของ การคืนหน่วยความจำ อย่างชัดแจ้ง จากโปรแกรมเมอร์ (ซึ่ง ถ้าทำไม่ถูกต้อง อาจทำให้เกิด dangling references) ในขณะที่ ณ เวลาเดียวกัน ขยะถูกเรียกคืน อัตโนมัติ สำหรับ การใช้ต่อไป

ระบบ ภาษา ซึ่ง ขยะถูกเรียกคืน อัตโนมัติ เรียกว่า กระทำการ **รวบรวมขยะ**

(Language systems that automatically reclaim garbage are said to perform garbage collection.)

โปรดสังเกตว่า การจัดการแบบกองซ้อน ของหน่วยความจำ ในสิ่งแวดล้อม ของภาษาเชิงโครงสร้างบล็อก สามารถ เรียกการรวบรวมขยะ ไว้เรียบร้อยแล้ว กล่าวคือ เมื่อ ออกจาก สโคป ของ การประกาศ ตัวแปรอัตโนมัติ สิ่งแวดล้อม จะเรียกคืน ตำแหน่งซึ่งจัดสรร ให้กับ ตัวแปรนั้น โดย "popping" หน่วยความจำ ซึ่งจัดสรร ให้ตัวแปร

ในอดีต ระบบภาษาเชิงหน้าที่ (functional language systems) โดยเฉพาะ ระบบ LISP การรวบรวมขยะ ดังเดิม เป็นวิธี ของ การจัดการคืนการจัดสรร ของ หน่วยความจำ ณ เวลาดำเนินงานจริงๆ แล้ว ในภาษา LISP การจัดสรรทั้งหมด เช่นเดียวกับ การคืนการจัดสรร ถูกกระทำ อย่างอัตโนมัติ

ระบบภาษาเชิงวัตถุ (Object-oriented language system) บ่อยครั้ง อยู่ที่ ตัวรวบรวมขยะ สำหรับ การเรียกคืน ของ หน่วยความจำ ระหว่าง การกระทำการ โปรแกรม สิ่งนี้ คือ กรณีของภาษา Smalltalk, Simula67, และ Eiffel (ภาษา C++ เป็น exception ที่สังเกตได้ ซึ่งกฎการจัดสรรและการคืนการจัดสรร ของ C ถูกเก็บไว้)

การออกแบบภาษา เป็น ปัจจัยสำคัญ (key factor) ตรงที่ สิ่งแวดล้อมเวลาคำเนินงาน ชนิดอะไร ซึ่งจำเป็น สำหรับ การกระทำการที่ถูกต้อง ของโปรแกรม ไม่เพียงเท่านั้น การออกแบบภาษา โดยตัวมันเอง ไม่ได้กล่าวชัดแจ้งว่า การจัดสรรหน่วยความจำ ชนิดใด ที่ต้องการ ตัวอย่างเช่น บทนิยาม ของ ภาษา Algol60 แนะนำเรื่อง โครงสร้างบล็อก และสนับสนุน การใช้สิ่งแวดล้อมแบบกองซ้อน โดยนัย และไม่ได้อธิบาย อย่างชัดแจ้ง

บทนิยาม ของ ภาษา LISP ไม่ได้กล่าวถึง การรวบรวมขยะ ถึงแม้ว่า ระบบ LISP โดยปกติ ไม่สามารถกระทำการได้อย่างมีเหตุผล ถ้า ไม่มีการรวบรวมขยะ

อีกวิธีหนึ่ง สำหรับ นักออกแบบภาษา ของภาษาเหมือน Algol เพื่อ ระบุ ความจำเป็น สำหรับ การรวบรวมขยะโดยอัตโนมัติ ให้รวม โปรซีเดอร์ **new** สำหรับตัวชี้ อยู่ในบทนิยามของภาษา แต่ใช้ไม่ได้ (fail) สำหรับ การรวม โปรซีเดอร์ **dispose** ที่สมนัย ภาษา Simula67 เข้าถึงโดยวิธีนี้ เช่นเดียวกับภาษา Ada

ใน Ada ตัวแปรชนิดตัวชี้ (pointer variable) เรียกว่า **ตัวแปร access** และนิยามดังนี้

```
type   intptr is access  INTEGER;
x : intptr;
```

ภาษา Ada การประกาศ ของ x จะกำหนด แต่แรกโดยอัตโนมัติ ให้เป็น **null** และวัตถุถูกชี้โดย x จะถูกจัดสรร ด้วย ข้อความสั่ง

```
x := new  intptr;
```

การคืนการจัดสรร ของ x ไม่สามารถกระทำได้ด้วยมือ อย่างไรก็ตาม ถ้าเรา ไม่ได้ประกาศ ชนิดของ x อย่างชัดแจ้งภายใต้การควบคุม ของ โปรแกรมเมอร์ ใน compiler directive

```
PRAGMA  CONTROLLED(intptr);
```

จากนั้น เราต้อง นำเข้า generic pocedure UNCHECKED_DEALLOCATION และทำให้มันใช้ intptr ดังนี้

```
procedure  Dispose intptr is new
        UNCHECKED_DEALLOCATION(INTEGER, intptr);
```

ขณะนี้ x จะถูกคืนการจัดสรร ด้วยข้อความสั่ง

```
Dispose Intptr(x);
```

ดังนั้น การออกแบบ ภาษา Ada สนับสนุนการใช้ ตัวรวบรวมขยะ ขณะเดียวกัน ก็

ยินยอมให้ โปรแกรมเมอร์ ใช้ ยุทธวิธี ของการคืนการจัดสรรด้วยมือ ซึ่งส่วนมากใช้กับ ภาษา Pascal หรือ C

5.7 การประเมินผลนิพจน์ (Expression Evaluation)

ภาษาโปรแกรม บ่อยครั้ง แยกความแตกต่าง ระหว่าง **นิพจน์** (expressions) **และข้อความสั่ง** (statements) ดังนี้

นิพจน์ ในรูปแบบ บริสุทธิ์ ของมัน ส่งกลับ ค่าหนึ่งค่า และ ไม่มีผลกระทบข้างเคียง นั่นคือ ไม่มีการเปลี่ยนแปลงใดๆ ใน หน่วยความจำ ของ โปรแกรม

(Expressions, in their pure form, return a value and produce no side effects, that is, no change to program memory.)

ในทางตรงกันข้าม ข้อความสั่ง จะถูกกระทำการ สำหรับ ผลกระทบข้างเคียง ของมัน และไม่ส่งกลับค่าใดๆ

(Statements, on the other hand, are executed for their side effects and return no value.)

บางครั้ง นิพจน์ มีผลกระทบข้างเคียง ได้ เช่นเดียวกัน ค่าส่งกลับ ซึ่งภาษาเช่นนี้ เรียกว่า **ภาษาเชิงนิพจน์** (expression languages) ซึ่ง ตัวสร้างภาษา ส่งกลับค่าต่างๆ และกระทำการ ทิ้งค่าเหล่านี้ และกระทำการ ผลกระทบข้างเคียง ทั้งคู่ ได้แก่ภาษา C, Algol68, และภาษาเชิงหน้าที่ เช่น LISP เป็นภาษาเชิงนิพจน์ด้วย

<u>ตัวอย่าง</u> ภาษา C ข้อความสั่งกำหนดค่า

```
x = y
```

ส่งกลับค่าของ y ดังนั้น

```
x = (y = z)
```

หมายถึง การกำหนดค่า ของ z ให้กับ x และให้กับ y ด้วย

<u>ตัวอย่าง</u> นิพจน์เดียวกัน แต่เขียนในภาษา Algol68 ดังนี้

```
x := (y := z)
```

โปรดสังเกตว่า ในภาษาเชิงนิพจน์นั้น การกำหนดค่า ถูกพิจารณา ให้เป็น ตัวดำเนินการแบบทวิภาค คล้ายกับ ตัวดำเนินการคำนวณ ในกรณีเช่นนั้น การทำก่อนของมัน ปกติ ต่ำกว่า ตัวดำเนินการคำนวณทั้งหมด และกระทำในลักษณะสลับที่แบบขวา (right associative) ดังนั้น

```
x = y = z + w
```

ในภาษา C หมายถึง กำหนดค่าของ ผลบวกของ z และ w ให้กับ x และ y ทั้งคู่

นิพจน์ ประกอบด้วย ตัวดำเนินการ และตัวถูกดำเนินการ

(Expressions contain operators and operands.)

ตัวดำเนินการ หมายถึง ฟังก์ชัน และตัวถูกดำเนินการ หมายถึง อาร์กิวเมนต์

(Operators are functions, and operands are arguments.)

ตัวดำเนินการ อาจจะเป็น **predefined (builtin)** หรือ **user defined**

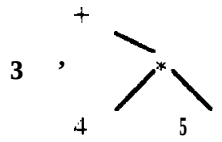
ตัวดำเนินการ สามารถ กระทำกับ ตัวถูกดำเนินการ หนึ่งตัว หรือ มากกว่าหนึ่งตัว

ตัวดำเนินการ ซึ่ง กระทำกับ ตัวถูกดำเนินการ หนึ่งตัว เรียกว่า **ตัวดำเนินการแบบเอกภาพ (unary operator)**

ตัวดำเนินการ ซึ่ง กระทำกับ ตัวถูกดำเนินการ สองตัว เรียกว่า **ตัวดำเนินการแบบทวิภาค (binary operator)**

ตัวดำเนินการ สามารถเขียน ด้วยสัญกรณ์เติมกลาง เติมหลัง หรือเติมหน้า (infix, postfix, or prefix notation) ซึ่งสมนัยกับ การแวะผ่านแบบตามลำดับ แบบหลังลำดับ หรือแบบก่อนลำดับ (inorder, postorder, or preorder traversal) ของ ต้นไม้วากยสัมพันธ์ ของนิพจน์คำนวณ

ตัวอย่าง นิพจน์เติมกลาง 3 + 4 * 5 วาดรูปต้นไม้วากยสัมพันธ์ ดังนี้



ซึ่งเขียนด้วย รูปแบบเติมหลัง เป็น 3 4 5 * + และรูปแบบเติมหน้า เป็น + 3 * 4 5

ข้อดี ของ รูปแบบเติมหลัง และรูปแบบเติมหน้า คือ ไม่จำเป็นต้อง มีเครื่องหมายวงเล็บ เพื่อแสดง อันดับ (order) ของตัวดำเนินการ ซึ่งจะถูกประยุกต์ใช้ ดังนั้น การทำก่อนของตัวดำเนินการ จะไม่กำกวม สำหรับ นิพจน์ซึ่งไม่มีวงเล็บกำกับ

ตัวอย่าง (3 + 4) * 5

เขียนด้วยรปแบบเติมหลัง เป็นดังนี้ 3 4 + 5 *

และรูปแบบเติมหน้า ดังนี้ * + 3 4 5

การสลับที่ (associativity) ของตัวดำเนินการ ถูกแสดงให้เห็นโดยตรง ในรูปแบบเติมหลัง และรูปแบบเติมหน้า โดยไม่จำเป็นต้องมีกฎ

ตัวอย่าง

นิพจน์เติมหลัง 3 4 5 + + เป็นการสลับที่แบบขวา และ 3 4 + 5 + เป็นการสลับที่แบบซ้าย เกี่ยวข้องกับ นิพจน์เติมกลาง 3 + 4 + 5

ภาษาโปรแกมส่วนใหญ่ ใช้ รูปแบบเติมหน้า สำหรับ predefined unary operators และใช้รูปแบบเติมกลาง สำหรับ predefined binary operators การสลับที่ตามที่กำหนดไว้ และกฎการทำก่อน

อย่างไรก็ตาม user-defined ฟังก์ชัน ในภาษาเหล่านี้ บ่อยครั้งที่เขียนในรูปแบบเติมหน้า

ตัวอย่าง ฟังก์ชัน Pascal ที่ประกาศดังนี้

```
function add(x, y : integer) : integer;
```

ต้องถูกประยุกต์ใช้ ตามวากยสัมพันธ์ ดังนี้

```
c := add(a, b);
```

ไม่สามารถเขียนเป็น c := a add b เช่นที่ predefined function "+" กระทำ

ภาษาโปรแกรมส่วนน้อย เช่น Ada และ Algol68 ยอมให้ สิ่งอำนวยความสะดวก จำนวนจำกัด สำหรับประกาศ ฟังก์ชัน ซึ่งนิยามในรูปแบบเติมกลาง ดังนี้

```
function "*" (a, b : MATRIX) return MATRIX IS
   . . .
```

ประกาศว่า ผลคูณของเมทริกซ์ ใน Ada เป็น ตัวดำเนินการแบบเติมกลาง (infix operator) ซึ่งมีสัญลักษณ์ เหมือนกับ ที่ใช้ในการคูณปกติ และ สามารถนำไปประยุกต์ใช้กับ วากยสัมพันธ์ปกติ ของ "*"

ตัวอย่าง

```
declare a, b, c : MARIX;


c := a * b:
```

ส่วน ภาษาเชิงวัตถุ (object-oriented languages) เช่น ภาษา Smalltalk 66% C++ มีกลไก ซึ่งยอมให้ เขียน user-defined functions ในรูปแบบเติมกลางได้

อย่างไรก็ตาม ภาษา LISP ใช้รูปแบบเติมหน้า พ้องกัน ทั้ง predefiued และ user-defined functions ทั้งนี้ นิพจน์ ต้องใส่วงเล็บอย่างบริบูรณ์ (fully-parenthesized) นั่นคือ ตัวดำเนินการ และตัวถูกดำเนินการ ทั้งหมดต้องอยู่ภายในวงเล็บ ที่เป็นเช่นนี้ เพราะว่า ตัวดำเนินการ ของ ภาษา

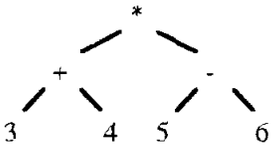
LISP สามารถกระทำ กับ อาร์กิวเมนต์ ซึ่งเป็น ตัวถูกดำเนินการ จำนวน กี่ตัว ก็ได้ ดังนั้น นิพจน์ 3 + 4 * 5 และ นิพจน์ (3 + 4) * 5 ในภาษา LISP จะเขียนดังนี้

(+ 3 (* 4 5))

และ (* (+ 3 4) 5)

ภาษาโปรแกรม แต่ละภาษา จะมีกฎ สำหรับ การประเมินผลนิพจน์ กฎการประเมินผล ที่ใช้กันส่วนใหญ่ นั้น ตัวถูกดำเนินการทั้งหมด ถูกประเมินผล เป็นอันดับแรก จากนั้น applied ตัวดำเนินการ ให้กับ ตัวถูกดำเนินการ สิ่งนี้เรียกว่า **การประเมินผลแบบอันดับเชิงหน้าที่** (applicative order evaluation) หรือ บางครั้ง เรียกว่า **การประเมินผลแบบเข้มงวด** (strict evaluation) และเป็นกฎที่ใช้กันส่วนใหญ่ ในภาษาโปรแกรมต่างๆ ซึ่งสมนัยกับ การประเมินผล แบบจากล่างขึ้นบน (a bottom-up evaluation) ของค่า ณ โหนดต่างๆ ของ ต้นไม้วากยสัมพันธ์ ซึ่งแทนนิพจน์นั้น

ตัวอย่าง นิพจน์ (3 + 4) * (5 - 6) ถูกแทนด้วย ต้นไม้วากยสัมพันธ์ ข้างล่างนี้



ในการประเมินผล แบบอันดับเชิงหน้าที่ สิ่งแรก โหนด "+" และ โหนด "-" ประเมินผลแล้วให้ผลลัพธ์เป็น 7 และ -1 ตามลำดับ จากนั้น applied ด้วย โหนด "*" ได้ผลลัพธ์เป็น -7

จงพิจารณา user-defined function ซึ่งเขียนด้วยรูปแบบเติมหน้า ใน วากยสัมพันธ์ของ Pascal ของนิพจน์ข้างล่างนี้

times(plus(3, 4), minus (5, 6))

อันดับเชิงหน้าที่ กล่าวว่า สิ่งแรก ประเมินผล อาร์กิวเมนต์ ดังนั้นเรียก plus(3, 4) และ minus(5, 6) ซึ่ง ประเมินผลโดย ใช้อันดับเชิงหน้าที่ จากนั้น การเรียก ถูก แทนที่ด้วย ค่าส่งกลับของมัน ซึ่งได้แก่ 7 และ -1 สุดท้ายเรียก

times(7, -1)

ให้กระทำ

คำถามคือ นิพจน์ย่อย (3 + 4) และ (5 - 6) ซึ่งจะถูกคำนวณ เรียงลำดับกันอย่างไร? หรือ การเรียก plus(3, 4) และ minus(5, 6) มีการเรียงลำดับ ให้กระทำชุดไหนก่อนหลัง

การเรียงลำดับแบบธรรมชาติ คือ จากซ้ายไปขวา (left to right) สมนัย กับ การแวะผ่าน

แบบจากซ้ายไปขวา ของต้นไม้วากยสัมพันธ์ อย่างไรก็ตาม ภาษาโปรแกรม จำนวนมาก กล่าวไว้ชัดแจ้งว่า ไม่มีการกำหนด ลำดับของ การประเมินผลอาร์กิวเมนต์ ให้กับ user-defined functions และ สิ่งนี้ บางครั้งเป็นจริง เช่นเดียวกับ predefined operators

มีเหตุผลหลายข้อ สำหรับ คำกล่าว ข้างต้นนี้

ข้อแรก เครื่องคอมพิวเตอร์ ต่างชนิดกัน อาจมีความต้องการ ที่แตกต่างกัน สำหรับ โครงสร้าง การเรียก โปรซีเดอร์ และ ฟังก์ชัน

ข้อสอง ตัวแปลภาษา อาจจะพยายาม จัดเรียง (rearrange) ลำดับ ของ การคำนวณ เพื่อให้ มีประสิทธิภาพมากขึ้น ตัวอย่างเช่น จงพิจารณา นิพจน์ (3 + 4) * (3 + 4) ตัวแปลภาษาอาจค้นพบว่า เป็นนิพจน์ย่อยเหมือนกัน คือ (3 + 4) ถูกใช้สองครั้ง และจะประเมินผลมันเพียงครั้งเดียวเท่านั้น

และในการประเมินผล การเรียกเช่น max(3, 4 + 5) ตัวแปลภาษา อาจจะประเมินผล 4 + 5 ก่อน 3 เพราะว่า การประเมินผลนิพจน์ ซึ่งซับซ้อนมากกว่า เป็น อันดับแรก จะมีประสิทธิภาพมากกว่า

ข้อที่สาม ถ้าการประเมินผลของนิพจน์ ไม่เกิด side effects แล้ว นิพจน์นั้น จะให้ผลลัพธ์ เหมือนกัน โดยไม่สนใจ การเรียงอันดับของการประเมินผล นิพจน์ย่อยของมัน ในทำนองเดียวกัน ถ้าการประเมินผล ของ อาร์กิวเมนต์ ของการเรียกฟังก์ชัน หรือ การเรียกโปรซีเดอร์ ไม่เกิด side effects แล้ว อันดับของการประเมินผล จะไม่สำคัญ เช่นกัน

อย่างไรก็ตาม ในกรณีที่มี side effects อันดับของการประเมินผล ทำให้มีความแตกต่าง จงพิจารณา ตัวอย่างโปรแกรมข้างล่างนี้

ตัวอย่าง โปรแกรม ภาษา Pascal

```
program sideEffect;
var   x : integer;
function f : integer;
begin
    x := x + 1;
    f := x;
end;
function p(a, b : integer) : integer;
```

```
begin
    p := b + a;
end,
begin
    x := 1;
    writeln(p(f, x)):
end.
```

ถ้าอาร์กิวเมนต์ ของ การเรียก p ใน ข้อความสั่ง writeln ถูกประเมินผลจากซ้ายไปขวา โปรแกรมนี้ ผลลัพธ์ พิมพ์ 4

ถ้าอาร์กิวเมนต์ ถูกประเมินผลจากขวาไปซ้าย โปรแกรมนี้ ผลลัพธ์จะพิมพ์ 3 เหตุผลคือ การเรียก ฟังก์ชัน f มี side effect เพราะว่ามันเปลี่ยนแปลง ค่าของ ตัวแปรส่วนกลาง x

ในทำนองเดียวกัน โปรแกรมภาษา C

```
int  f(int a, int b)
{return b+a;}

void main(void)
{int x, y;
x = 1;
y = f(x, x = x + 1);}
```

ถ้า อาร์กิวเมนต์ ของ f ถูกประเมินผล จากซ้ายไปขวา ตัวแปร y จะมีค่าเป็น 3

ถ้า ผลบวก ประเมินผล จากขวาไปซ้าย ตัวแปร y จะมีค่าเป็น 4

ในภาษาโปรแกรม ซึ่งกล่าวชัดแจ้งว่า ลำดับของการประเมินผล ของ อาร์กิวเมนต์ และ/หรือ นิพจน์ ไม่ได้กำหนดให้ โปรแกรม ซึ่งขึ้นอยู่กับ ลำดับของการประเมินผล สำหรับ ผลลัพธ์ จะไม่ถูกต้อง ถึงแม้ว่า พฤติกรรมของมัน สามารถคาดคะเนได้ สำหรับ ตัวแปลภาษา หนึ่งตัวหรือมากกว่าหนึ่งตัว

การประเมินผล ของ นิพจน์หนึ่งชุด บางครั้ง อาจจะกระทำได้ โดยไม่ต้องประเมินผล นิพจน์ย่อยของมัน ทั้งหมด กรณีที่น่าสนใจคือ นิพจน์แบบบลู หรือ นิพจน์เชิงตรรกะ

ตัวอย่าง นิพจน์แบบบลู

```
true or x
```

และ

```
x or true
```

ทั้งสองชุดข้างต้นนี้ เป็นจริงเสมอ ไม่ว่า x จะเป็นจริง หรือ เป็นเท็จ ในทำนองเดียวกัน นิพจน์ข้างล่างนี้

```
false and x
```

และ

```
x and false
```

เป็นเท็จเสมอ โดย ไม่ต้องสนใจค่าของ x

ในภาษาโปรแกรม เราอาจระบุว่า นิพจน์แบบบลู จะถูกประเมินผล ในลำดับ จากซ้ายไปขวา จนถึงจุด ซึ่งทราบค่าของ **ค่าความจริง** (truth value) ของนิพจน์ทั้งหมด จากนั้นจึงหยุดการประเมินผล ภาษาโปรแกรม ซึ่งใช้กฎเช่นนี้ เรียกว่า มีคุณสมบัติ **การประเมินผลแบบวงจรสั้น** (short-circuit evaluation) ของนิพจน์ แบบบลู หรือ นิพจน์แบบตรรกะ

การประเมินผลแบบวงจรสั้น มี ข้อดี หลายประการ คือ

ข้อที่หนึ่ง การทดสอบความถูกต้อง ของ ดรรชนี ของแถวลำดับ สามารถเขียน ใน นิพจน์เดียวกัน เป็นการทดสอบ ค่าดรรชนีล่างของมัน ตราบเท่าที่ การทดสอบพิสัย เกิดขึ้นครั้งแรก ดังนี้

```
if (i <= lastindex) and (a[i] = x) then . . .
```

โดยไม่ต้องทำการประเมินผลแบบวงจรสั้น การทดสอบ i <= lastindex ไม่ได้ป้องกัน ข้อผิดพลาด ซึ่งจะเกิดขึ้น ถ้า i > lastindex เพราะว่า นิพจน์ a[i] ในส่วนที่สอง ของ นิพจน์ จะยังคงถูกคำนวณ (สิ่งนี้ สมมติว่า ดรรชนีล่าง ของแถวลำดับ อยู่นอกพิสัย จะให้ ข้อผิดพลาด)

ถ้าไม่มี การประเมินผลแบบวงจร-สั้น การทดสอบ ต้องเขียน โดยใช้ nested if's ดังนี้

```
if (i <= lastindex) then if (a[i] = x) then . . .
```

ในทำนองเดียวกัน การทดสอบ nil pointer สามารถกระทำใน นิพจน์เดียวกัน เหมือนกับ dereference ของ pointer ตัวนั้น โดยใช้การประเมินผลแบบวงจรสั้น ดังนี้

```
if (p <> nil) and (p^.data = x) then . . .
```

โปรดสังเกตว่า ลำดับ ของการทดสอบ กลายเป็นสิ่งสำคัญ ในการประเมินผลแบบวงจรสั้น

การประเมินผลแบบวงจร-สั้น จะป้องกันเราจาก ข้อผิดพลาดเวลาดำเนินงาน ถ้าเราเขียน

ดังนี้

```
if (x <> 0) and (y mod x = 0) then  (* ok *)
```

แต่จะไม่ป้องกัน ถ้าเขียนดังนี้

```
if (y mod x = 0) and (x <> 0) then  (* not ok! *)
```

ในภาษา Pascal ตัวดำเนินการแบบบลู **and** และ **or** ไม่ใช่วงจร-สั้น (short-circuit)

แต่ในภาษา Modula-2 ตัวดำเนินการแบบบลู AND และ OR เป็นวงจร-สั้น

ภาษา C ตัวดำเนินการเชิงตรรกะ ที่สมนัยกัน "&&" และ "||" เป็น วงจร-สั้น

ภาษา Ada มีตัวดำเนินการแบบบลู ทั้งวงจร-สั้น และ วงจร-ไม่สั้น (non-short-circuit)

ดังนี้

ตัวดำเนินการ **and** และ ตัวดำเนินการ **or** ไม่ใช่ วงจร-สั้น ในขณะที่ ตัวดำเนินการ ซึ่งสมนัยกับ ตัวดำเนินการ แบบวงจร-สั้น เขียนเป็น **and then** และ **or else** ดังนี้

```
if (i <= lastindex) and then (a(i) = x) then . . .
```

อย่างเป็นทางการ เราสามารถนิยาม ตัวดำเนินการแบบบลู วงจร-สั้น (*short-circuit Boolean operators*) โดยใช้ ตัวดำเนินการแบบบลู **if-then-else** ดังนี้

ตัวดำเนินการ if-then-else มีอาร์กิวเมนต์ สามตัว และเขียนดังนี้

```
if  a  then  b  else  c
```

อรรถศาสตร์ (semantics) คือ อันดับแรก ประเมินผล a ถ้าค่าของ a เป็นจริง แล้ว b จะถูกประเมินผล และ ค่าของ b คือค่าของนิพจน์

ถ้าค่าของ a เป็นเท็จ จากนั้น c จะถูกประเมินผล และค่าของ c คือค่าของนิพจน์

ขณะนี้ เราสามารถ นิยาม ตัวดำเนินการแบบบลู วงจร-สั้น ซึ่งเรียกว่า **cand** และ **cor** (สำหรับ เงื่อนไข and และเงื่อนไข or) เพื่อแยกความแตกต่าง ออกจาก ตัวดำเนินการ มาตรฐาน จะเป็นดังนี้

```
a  cand  b  =  if  a  then  b  else  false
a  cor  b   =  if  a  then  true  else  b
```

ตัวดำเนินการแบบบลู วงจร-สั้น เป็นกรณีพิเศษของ ตัวดำเนินการซึ่ง ทำให้ การประเมินผล อาร์กิวเมนต์ของมัน **ช้าลง** (delay)

สถานการณ์โดยทั่วไป เรียกว่า **การประเมินผลแบบช้าลง** (delayed evaluation)

ในกรณีของ ตัวดำเนินการแบบ วงจร-สั้น a cand b และ a cor b ทั้งสองแบบนี้

การประเมินผล ของ b จะช้าออกไป จนกว่า a จะถูกประเมินผล

ในทำนองเดียวกัน ตัวดำเนินการ if-then-else แบบบลู delays การประเมินผล ของทั้ง b และ c จนกระทั่ง a ถูกประเมินผล

รูปแบบอย่างหนึ่งของการประเมินผล แบบทำให้ช้าลง ซึ่งมีความสำคัญเชิงทฤษฎี และในเชิงปฏิบัติ เช่นเดียวกัน โดยเฉพาะ สำหรับ ภาษาโปรแกรมเชิงหน้าที่ คือ **การประเมินผลแบบอันดับปกติ** (normal order evaluation) ซึ่งการประเมินผล ของอาร์กิวเมนต์ แต่ละตัว หรือ นิพจน์ย่อยแต่ละตัว ถูกทำให้ช้าลง จนกระทั่ง มันมีความจำเป็นจริงๆ ในการคำนวณของผลลัพธ์

วิธีหนึ่ง ของการตีความหมาย กฎการประเมินผลแบบอันดับปกติ สำหรับ ฟังก์ชัน คือ **โดยการแทนที่ข้อความ** (textual substitution) ของ อาร์กิวเมนต์ ภายใน body ของฟังก์ชัน

ตัวอย่าง จงพิจารณา วากยสัมพันธ์ Pascal ข้างล่างนี้

```
function  sq(x : integer) : integer;
begin
    sq := x * x;
end;
```

ถ้า sq ใช้การประเมินผลแบบอันดับปกติ สำหรับอาร์กิวเมนต์ของมัน และถ้าเราเรียกฟังก์ชัน sq ดังนี้

```
sq(3 + 4)
```

จะมีผลลัพธ์ ของการแทนที่ นิพจน์ 3 + 4 ใน รหัส sq โดยไม่มีการประเมินผลมันเป็นอันดับแรก จะเป็นดังนี้

```
sq(3 + 4) = (3 + 4) * (3 + 4)
```

หลังจากการแทนที่ ข้างต้นนี้แล้วเท่านั้น จึงจะมีการประเมินผล 3 + 4 โปรดสังเกตว่าสิ่งนี้ แตกต่างจาก การประเมินผลแบบอันดับเชิงหน้าที่ ซึ่ง 3 + 4 จะถูกประเมินผล สองครั้ง แทนที่ จะเป็น หนึ่งครั้ง

แบบฝึกหัด (Exercises)

1. Assume the language Pascal, C, or Modula-2. Give as precise binding times as you can for the following attributes, and give reason for your answers :

   (a) the number of significant digits of a real number

   (b) the meaning of **char**

   (c) the size of an array variable

   (d) the size of an array parameter

   (e) the location of a local variable

   (f) the value of a constant

   (g) the location of a function

2. DISCUSS the meaning of the following statement : early binding promotes security and efficiency, while late binding promotes flexibility and expressiveness

3. In FORTRAN, global variables are created using a COMMON statement, but it is not required that the global variables have the same name in each subroutine or function. Thus, in the code

```
      FUNCTION  F
      COMMON  N

      END
      SUBROUTINE  S
      COMMON  M

      END
```

   Variable N in function F and variable M in subroutine S share the same memory, so they behave as different names for the same global variable.

(a) Compare this method of creating global variables to that of Pascal or C. How is it better'? How is it worse?

(b) Describe the difference between variables that are COMMON and variables that are EQUIVALENCE

4. In Pascal there is no distinction between global variables and those local to the main program. Is this distinction made in C? Modula-21 FORTRAN? Is such a distinction useful? Why or why not?

5. Execute the following Pascal program, and explain the resulting output :

```
program extent;
procedure p;
var y : integer:
begin
    writeln(y);
    y := 2;
end;
begin
    p: p:
end.
```

6. Describe the scopes of the declarations in the following Pascal program. How would the scopes change using dynamic instead of static scoping? What does the program print in each case?

```
program scope:
var a. b : integer:
```

```
function p : integer;
var a : integer;
begin
    a := 0; b := I: p := 2;
end;
procedure print;
begin
    writeln(a); writeln( b); writeln(p):
end;
procedure q;
var b. p : integer-:
begin
    a := 3; b := 4; p := 5: print;
end:
begin
    a := p:
    q:
end.
```

7. จงอธิบายความแตกต่างระหว่างสนนามและผลกระทบข้างเคียง
(Explain the difference between aliasing and side effects.)

8. A common definition of side effect is a change to a nonlocal variable or to the input or output made by a function or procedure. **Compare** this definition of side effect to the one in the text.

9. Given the following Pascal program, draw box-and-circle diagrams of the variables after the assignment to x^^. Which variables are aliases of each other at that point? What does the program print?

```
program alias;
type
    intptr = ^integer;
    ptrptr = ^intptr;
var
    x : ptrptr;
    y : intptr:
    z : integer;
begin
    new(x):
    new(y);
    z := 1;
    y^ := 2;
    x^ := y;
    x^^ := z:
    writeln(y^);
    z := 3;
    writeln(y^);
    x^^ := 4;
    writeln(z);
end.
```

10. Explain the reason for me two calls to new in the previous exercise. What would be printed if the call to new(x) were left out? The call to new(y)?

11. Pointers in a programming language are generally used with records to define recursive, dynamic data structures such as lists and trees. In these cases, pointers always point to records, as in the following declaration of a linked list of integers :

```
type link = ^listrec;
     listrec = record
                    data : integer;
                    next : link;
               end;
```

Suppose we decided to restrict the declaration of pointers to pointers to records. so that the declaration

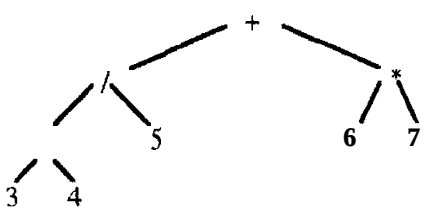
```
type ptrl = *integer;
```

is illegal but the declaration of link above remains legal. Discuss the advantages and disadvantages of such restriction.

12. Rewrite the following infix expression in prefix and postfix and draw the syntax tree :

    (3 - 4) / 5 + 6 * 7

ANSWER



prefix expression : + / - 3 4 5 * 6 7

postfix expression : 3 4 - 5 / 6 7 * +

13. Write a BNF description of

    a) postfix arithmetic expressions and

    b) prefix arithmetic expressions

ANSWER

a) <exp> ::= <exp><exp><op> | <number>

<op> ::= + | *

<number> ::= <number><digit> | <digit>

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

14. In LISP the following unparenthesized prefix expression is ambiguous :

+ 3 * 4 5 6

Why? Give two possible parenthesized interpretations.

ANSWER The problem is that the arithmetic operations can take more than two operands, so two interpretations of the expression are

(+ 3 (* 4 5 6)

and

(+ 3 (* 4 5) 6)

with values 123 and 29, respectively,

15. Test the following program on your FORTRAN compiler. Try to explain its behavior.

```
logical  f
if (.false. .and. (sqrt(-1.0) .gt. 0.0)) then
     print *, 'OK'

else
     print *, 'not-OK'
endif
if (.false. .and. f( )) then
     print *, "OK'
else
     print *, 'not-OK'
endif
if (f( ) .and. (sqrt(-1.0) .gt. 0.0)) then
```

```
        print *, 'OK'
    else
        prmt *, 'not-OK'
    endif
    end
    logical function f( )
        print *, 'in f!'
        f = .false.
    end
```

16. Some Pascal compilers use short-circuit evaluation of Booleans even though Standard Pascal defines the Boolean operators as non-short-circuit Write 3 program to test your Pascal compiler for short-circuit evaluation of both **and** and **or**