บทที่ 4

==========

การแปลภาษาและการประมวลผล
(Language translation and processing)

วัตถุประสงค์

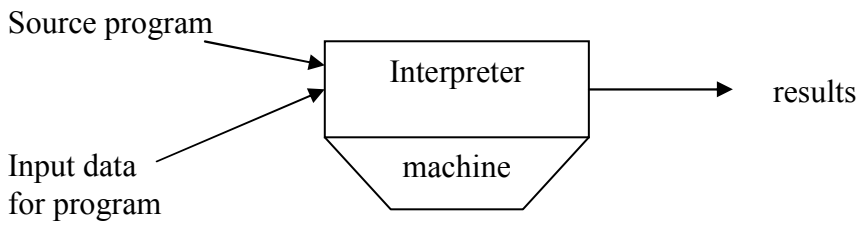
1) เพื่อให้ผู้เรียนสามารถแยกความแตกต่างระหว่างอินเตอร์พรีตเตอร์และคอมไพเลอร์
2) เพื่อให้ผู้เรียนเข้าใจถึงขั้นตอนต่างๆ ของคอมไพเลอร์ และสามารถอธิบายการทำงานของคอมไพเลอร์
3) เพื่อให้ผู้เรียนรู้จักส่วนประกอบและหน้าที่การทำงานของเครื่องจักรเสมือน
4) เพื่อให้ผู้เรียนรู้จักโครงสร้างของหน่วยความจำหลักที่มีส่วนเกี่ยวข้องกับการประมวลผลโปรแกรมคอมพิวเตอร์

ภาษาการทำโปรแกรมที่ใช้กันอยู่ปัจจุบันเช่น ภาษา C, Java เรียกว่าภาษาระดับสูง เนื่องจากรูปแบบคำสั่งต่างๆ ของภาษาอยู่ในระดับที่สูงกว่าความเข้าใจของเครื่องคอมพิวเตอร์ ภาษาที่เครื่องคอมพิวเตอร์เข้าใจและสามารถประมวลผลตามคำสั่งได้ทันทีจะเรียกว่าภาษาระดับต่ำ ซึ่งได้แก่ภาษาเครื่อง (machine language) ภาษาเครื่องมีข้อดีคือ สามารถประมวลผลได้ทันทีในเวลาที่รวดเร็ว แต่ข้อเสียที่สำคัญคือ รูปแบบคำสั่งที่มีแต่ตัวเลข 0 และ 1 มนุษย์เขียนและอ่านเพื่อทำความเข้าใจได้ยาก จึงต้องมีการสร้างภาษาระดับสูงที่อ่านและเขียนได้ง่ายขึ้น แต่ก็ต้องชดเชยด้วยการสร้างโปรแกรมพิเศษขึ้นมาช่วยในการแปลจากภาษาระดับสูงให้เป็นภาษาเครื่อง โปรแกรมที่ช่วยในการแปลมีอยู่สองประเภท คือ อินเตอร์พรีตเตอร์ และคอมไพเลอร์ โปรแกรมทั้งสองประเภททำหน้าที่ เปลี่ยนภาษาระดับสูงให้อยู่ในรูปแบบของภาษาเครื่อง ที่ฮาร์ดแวร์สามารถทำงานได้ทันที แต่รูปแบบการแปลในปัจจุบันนิยมแปลจากภาษาระดับสูงให้เป็นภาษาเครื่องของเครื่องจักรเสมือน (virtual machine) และเมื่อจะประมวลผลจริงจะแปลงจากภาษาเครื่องบนเครื่องจักรเสมือนให้เป็นภาษาเครื่องที่แท้จริง ทั้งนี้เพื่อประโยชน์ในการเคลื่อนย้าย (portable) โปรแกรมไปประมวลผลได้บนเครื่องหลายระบบ

4.1 อินเตอร์พรีตเตอร์และคอมไพเลอร์

(Interpreter and compiler)

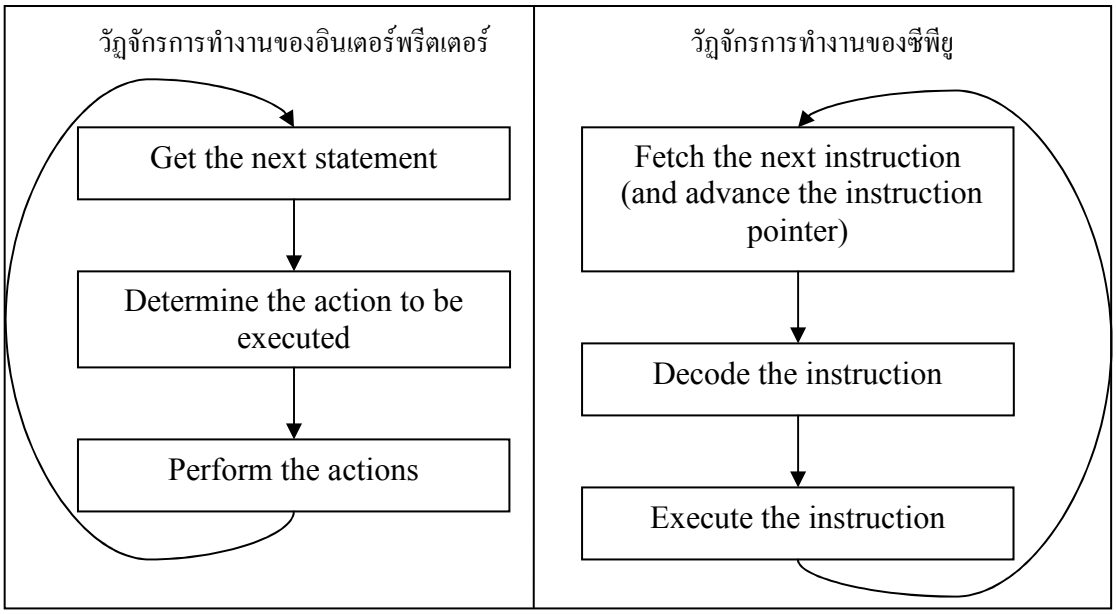
อินเตอร์พรีตเตอร์ คือโปรแกรมที่ทำหน้าที่แปลคำสั่งในภาษาระดับสูงให้เป็นภาษาเครื่องพร้อมทั้งสั่งการประมวลผลคำสั่ง (แสดงเป็นแผนภาพได้ดังรูปที่ 4.1) อินเตอร์พรีตเตอร์ จึงเป็นโปรแกรมพิเศษที่รับอินพุตเป็นอีกโปรแกรมเรียกว่า โปรแกรมต้นทาง (source program) และรับข้อมูล (data) ที่อาจต้องใช้ประกอบการทำงานของโปรแกรมต้นทาง



รูปที่ 4.1 โครงสร้างการทำงานของอินเตอร์พรีตเตอร์

กระบวนการแปลภาษาและการประมวลผลของอินเตอร์พรีตเตอร์ จะเริ่มต้นด้วยการอ่านคำสั่งในโปรแกรมต้นทางทีละคำสั่ง แปลความหมายของคำสั่งและสั่งการเครื่องให้ประมวลผลตามคำสั่งนั้น จากนั้นก็จะอ่านคำสั่งต่อไป, แปล, และสั่งการประมวลผลวนเวียนเป็นวัฏจักรไปเช่นนี้จนจบคำสั่งในโปรแกรมต้นทาง วัฏจักรการทำงานนี้จะสอดคล้องกับขั้นตอนการทำงานภายในของหน่วยประมวลผลในระบบคอมพิวเตอร์ ที่เรียกว่า วัฏจักรเครื่อง (machine cycle) ซึ่งประกอบด้วย การอ่านคำสั่ง (fetch) จากตำแหน่ง

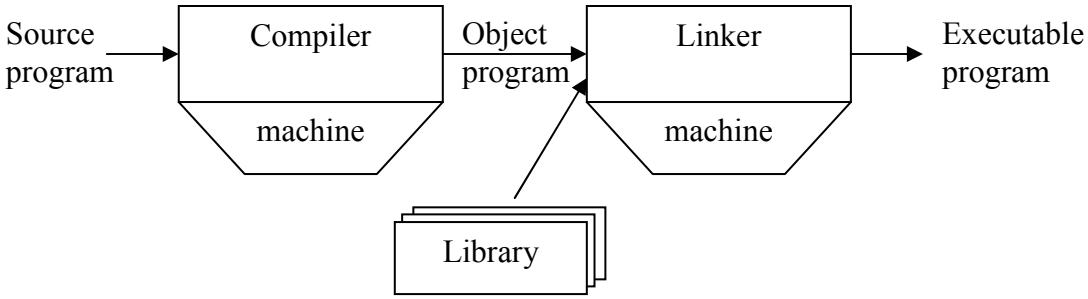
ที่ระบุโดย รีจิสเตอร์ตำแหน่งโปรแกรม (program address register or instruction pointer), การแปลรหัสคำสั่ง (decode), การทำงานตามคำสั่ง (execute) ความสอดคล้องของสองวัฏจักรนี้ แสดงเปรียบเทียบได้ดังรูปที่ 4.2



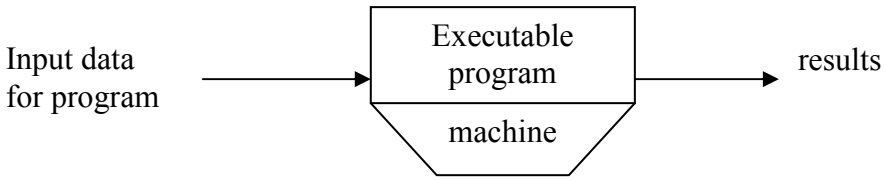
รูปที่ 4.2 เปรียบเทียบวัฏจักรการทำงานของอินเตอร์พรีตเตอร์และซีพียู

ในกรณีมีข้อผิดพลาดในโปรแกรมต้นทาง เช่นคำสั่งเขียนผิดไวยากรณ์ อินเตอร์พรีตเตอร์จะแสดงข้อความเตือนเหตุผิดพลาด (error message) และหยุดกระบวนการแปลจนกว่าผู้ใช้จะแก้ไขข้อผิดพลาดในโปรแกรมต้นทาง และสั่งให้อินเตอร์พรีตเตอร์เริ่มต้นทำงานใหม่

คอมไพเลอร์เป็นโปรแกรมที่ทำหน้าที่แปลคำสั่งในภาษาระดับสูง และสั่งการประมวลผลเหมือนกับอินเตอร์พรีตเตอร์ แต่มีข้อแตกต่างที่ขั้นตอนการทำงานซับซ้อนกว่า และแยกการแปลออกจากการสั่งการประมวลผล โครงสร้างการทำงานของคอมไพเลอร์แสดงได้ดังรูปที่ 4.3



(a) ขั้นตอนการแปลเป็นภาษาเครื่อง



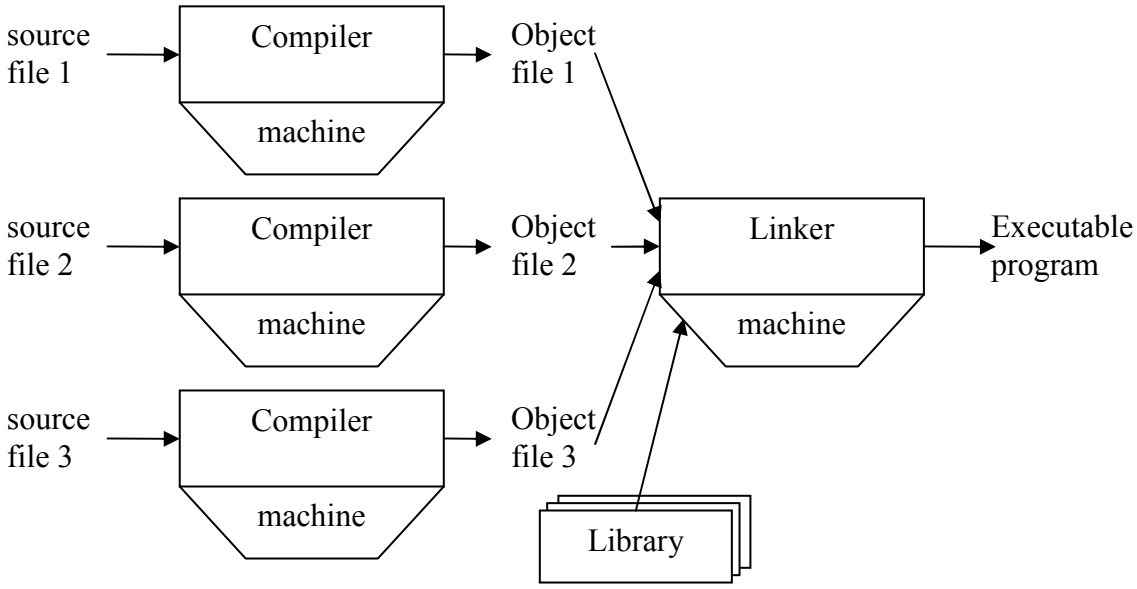
(b) ขั้นตอนการสั่งประมวลผลโปรแกรม

รูปที่ 4.3 โครงสร้างการทำงานของคอมไพเลอร์

กระบวนการทำงานของคอมไพเลอร์จะเริ่มต้นด้วยการรับอินพุตเป็นโปรแกรมต้นทาง แล้วแปลเป็นโปรแกรมจุดหมาย (object program) ซึ่งอยู่ในรูปแบบภาษาระดับต่ำ เช่น ภาษาแอสเซมบลี หรือภาษาเครื่อง แต่ยังไม่สามารถประมวลผลได้เพราะการแปลในขั้นตอนนี้ยังไม่ได้แปลงจากคำสั่งในภาษาระดับสูงไปเป็นคำสั่งทั้งหมดที่เทียบเท่ากันในรูปแบบภาษาเครื่อง เพียงแต่เป็นการแปลงไปสู่คำสั่งอ้างอิง (reference) ที่ระบุเป็นชื่ออ้างอิงไปถึงชุดคำสั่งภาษาเครื่องที่เก็บอยู่ในไลบรารี ตัวอย่างเช่นคำสั่ง write ในภาษาปาสคาล เมื่อผ่านการแปลในขั้นแรก คำสั่ง write จะถูกเปลี่ยนเป็น write_cmd ซึ่งเป็นชื่ออ้างอิงไปยังไฟล์ในไลบรารี ชื่อ pascal.lib ไฟล์นี้จะมีคำสั่งภาษาเครื่องที่เทียบเท่ากับคำสั่ง write ในภาษาปาสคาล การดึงชุดคำสั่งภาษาเครื่องมาแทนคำสั่ง write จะเกิดขึ้นในขั้นตอนการลิงค์โดยโปรแกรมที่เรียกว่า ตัวเชื่อมโยง (linker) หลังจากการเชื่อมโยงจึงจะได้โปรแกรมในรูปแบบภาษาเครื่องที่สมบูรณ์ เรียกว่า โปรแกรมพร้อมกระทำการ (executable program)

เมื่อจบจากขั้นตอนการแปล เราจะได้โปรแกรมในรูปแบบภาษาเครื่องที่สามารถบันทึกเก็บไว้ได้ เมื่อต้องการจะสั่งประมวลผลเราจะเรียกใช้โปรแกรมพร้อมกระทำการ โดยป้อนข้อมูลเข้าให้กับโปรแกรม ประโยชน์ของการมีโปรแกรมพร้อมกระทำการคือ เราสามารถสั่งประมวลผลโปรแกรมได้หลายครั้งกับข้อมูลเข้าต่างๆ กัน โดยไม่ต้องเสียเวลากับขั้นตอนการแปลซ้ำอีกครั้ง ซึ่งจะแตกต่างจากอินเตอร์พรีตเตอร์ที่การประมวลผลจะต้องเริ่มที่การอ่านคำสั่ง-แปลคำสั่ง-ปฏิบัติตามคำสั่ง เป็นวัฏจักรเช่นนี้เสมอ ทั้งนี้เนื่องจากอินเตอร์พรีตเตอร์ไม่มีการบันทึกโปรแกรมพร้อมกระทำการที่อยู่ในรูปแบบภาษาเครื่องเก็บไว้ จึงต้องแปลคำสั่งใหม่ทุกครั้งที่มีการสั่งประมวลผลโปรแกรมต้นทาง แต่ข้อเด่นของอินเตอร์พรีตเตอร์ที่เหนือกว่าคอมไพเลอร์ คือ โครงสร้างการทำงานไม่ซับซ้อน ทำให้สามารถพัฒนาอินเตอร์พรีตเตอร์มาใช้ช่วยในการ

แปลและประมวลผลโปรแกรมได้ในเวลาที่รวดเร็วกว่าการพัฒนาคอมไพเลอร์ ที่มีขั้นตอนมาก และซับซ้อนยุ่งยาก แต่เมื่อพัฒนาได้สำเร็จคอมไพเลอร์จะทำงานได้เร็วกว่า และสะดวกกว่าด้วยวิธีการแปลแบบที่เรียกว่า การแปลแยกส่วน (separate compilation) โดยโปรแกรมต้นทางสามารถถูกแยกเป็นโมดูลย่อยๆ ไว้ในแต่ละไฟล์ แล้วแยกแปลแต่ละไฟล์ได้เป็นโปรแกรมจุดหมาย หรือ อ็อบเจ็คไฟล์ย่อยๆ (ดังรูปที่ 4.4) จากนั้นตัวเชื่อมโยงจะทำหน้าที่รวมอ็อบเจ็คไฟล์ และชุดคำสั่งภาษาเครื่องจากไลบรารีสร้างเป็นโปรแกรมพร้อมกระทำการ หรือบางครั้งเรียกว่า ไบนารีไฟล์ (binary file)



รูปที่ 4.4 การแปลแบบแยกส่วนของคอมไพเลอร์

ในกรณีที่โปรแกรมต้นทางมีการใช้แมโคร (macro) ตั้งชื่อเพื่อแทนค่าหรือคำสั่ง ด้วยเจตนาที่จะให้คำสั่งในโปรแกรมอ่านได้ง่ายและสื่อความหมายดีขึ้น ตัวอย่างเช่น ข้อความสั่งในภาษา C ต่อไปนี้ที่มีการใช้แมโครด้วยการเริ่มคำสั่งด้วย *#define* เพื่อนิยามชื่อ *UPPER_LIMIT* ให้หมายถึงค่า 100 และนิยามแมโครชื่อ *INCREMENT* ให้หมายถึงคำสั่งเพิ่มค่าขึ้นหนึ่งค่า โดยแมโครนี้มีการรับพารามิเตอร์ว่าจะให้เพิ่มค่ากับตัวแปรใด

```
#define UPPER_LIMIT          100
#define INCREMENT(a)         a++
...
int n, sum=0;
for(n=1; n<=UPPER_LIMIT; INCREMENT(n) )
        {
                sum +=n;
        }
```

การแปลโปรแกรมต้นทางที่มีการใช้แมโคร จะมีขั้นตอนเพิ่มมากขึ้นกว่าปกติ นั่นคือจะต้องเปลี่ยนชื่อแมโครให้เป็นค่าหรือคำสั่ง จากตัวอย่างภาษา C ข้างต้น เมื่อผ่านขั้นตอนการแปลเบื้องต้น จะได้ข้อความสั่งต่อไปนี้

```
...
int n, sum=0;
for (n = 1; n <= 100; n++)
        {
                sum += n;
        }
```

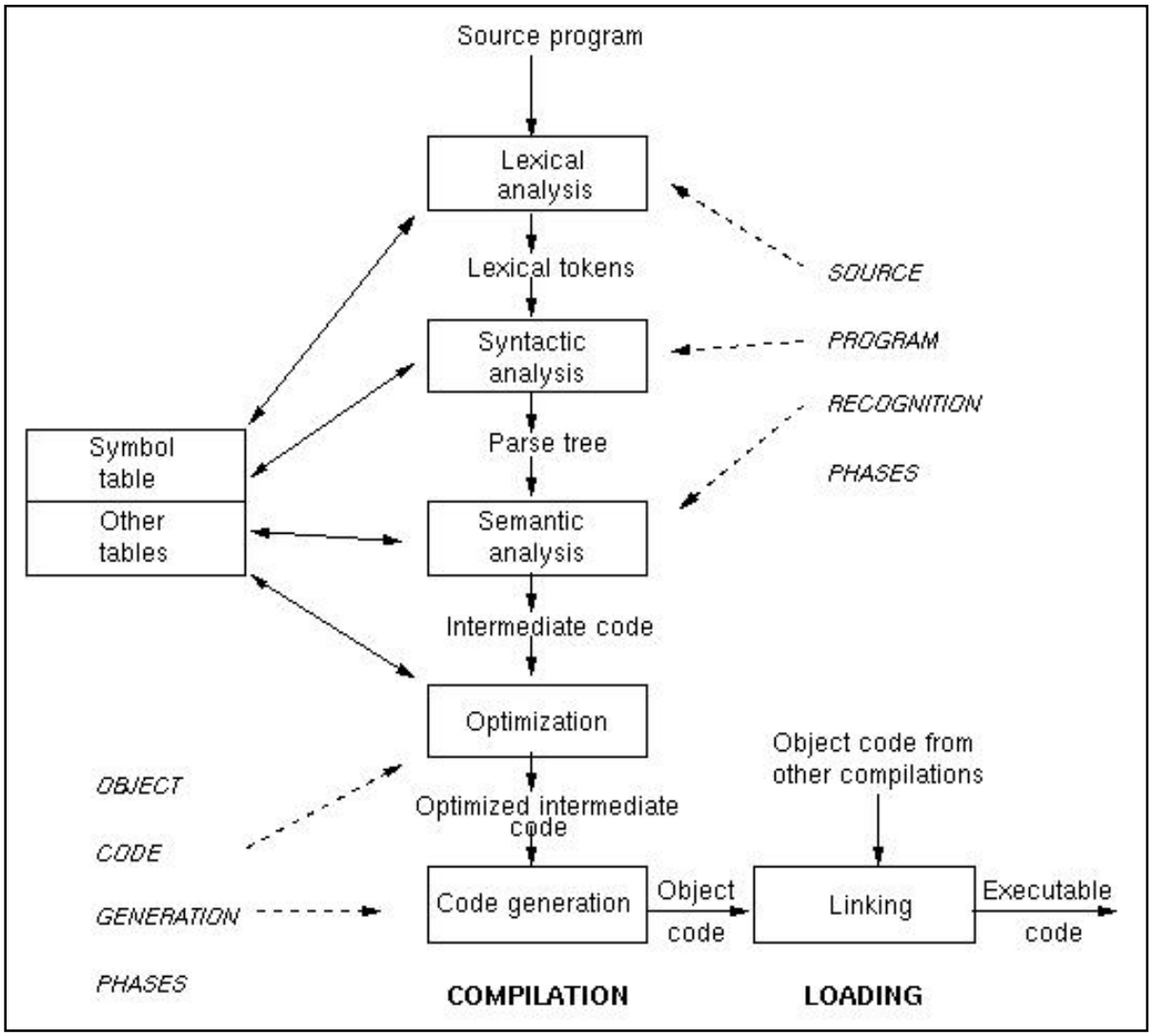
ขั้นตอนการแปลเบื้องต้น (pre-processing) นี้มีชื่อเรียกเฉพาะว่า การประมวลผลแมโคร (macro processing) และโปรแกรมที่ทำหน้าที่นี้เรียกว่า โปรแกรมประมวลผลแมโคร (macro processor)

จะเห็นได้ว่าการแปลโปรแกรมต้นทางที่เขียนอยู่ในภาษาระดับสูงให้เป็นภาษาที่เครื่องเข้าใจและประมวลผลได้ จะต้องใช้โปรแกรมช่วยงานจำนวนมาก เช่น เอดิเตอร์, คอมไพเลอร์, ลิงเกอร์, ดีบักเกอร์, แมโครโพรเซสเซอร์ โปรแกรมเหล่านี้เรียกว่าโปรแกรมช่วยงาน (tools) หรือมีชื่อเรียกโดยรวมว่า สภาวะแวดล้อมในการทำโปรแกรม (programming environments) ในระบบปฏิบัติการยูนิกซ์ โปรแกรมช่วยงานเหล่านี้จะแยกกันอยู่ ขึ้นอยู่กับผู้ใช้จะเรียกใช้ แต่ในเครื่องส่วนบุคคลที่ใช้ระบบปฏิบัติการวินโดส์ โปรแกรมช่วยงานทั้งหลายมักจะถูกรวบรวมไว้ด้วยกัน เรียกว่า Integrate Development Environment หรือ IDE เช่น Microsoft Visual C/C++, Java Development Kit

4.2 ขั้นตอนการทำงานของคอมไพเลอร์

(Stages in compilation)

โครงสร้างการทำงานของคอมไพเลอร์จะประกอบด้วยช่วงการทำงานหลักสองช่วง คือ ช่วงวิเคราะห์ตรวจสอบคำสั่งในโปรแกรมต้นทาง และช่วงแปลเป็นโปรแกรมจุดหมาย หรือเรียกว่า การสร้างรหัสจุดหมาย (object code generation) ในแต่ละช่วงของการทำงานยังแบ่งออกเป็นขั้นตอนย่อยๆ ดังแสดงในรูปที่ 4.5



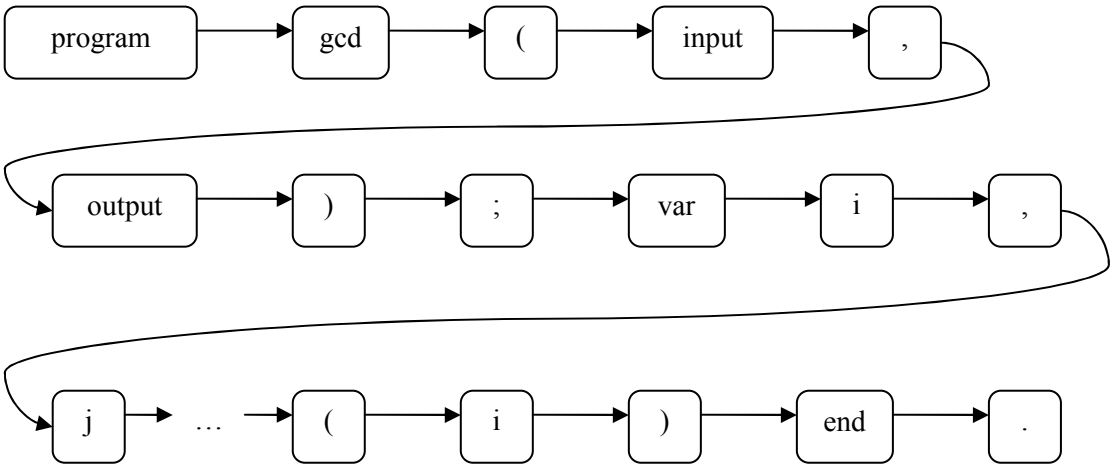
รูปที่ 4.5 โครงสร้างแสดงขั้นตอนการทำงานของคอมไพเลอร์

<u>ขั้นตอนที่ 1 การวิเคราะห์ศัพท์</u> (Lexical analysis or scanning)

เป็นขั้นตอนแรกสุดของคอมไพเลอร์ที่ทำหน้าที่อ่าน หรือสแกนคำสั่งทั้งหมดในโปแกรมต้นทาง เพื่อแยกคำและเครื่องหมายต่างๆ คำแต่ละคำหรือเครื่องหมายแต่ละตัวจะเรียกว่าโทเค็น (token) ตัวอย่างเช่น โปรแกรมต้นทางในรูปแบบภาษาปาสคาล ตามรูปที่ 4.6 เมื่อถูกสแกนและแยกเป็นโทเค็นจะได้สายของโทเค็น (token stream) ดังรูปที่ 4.7

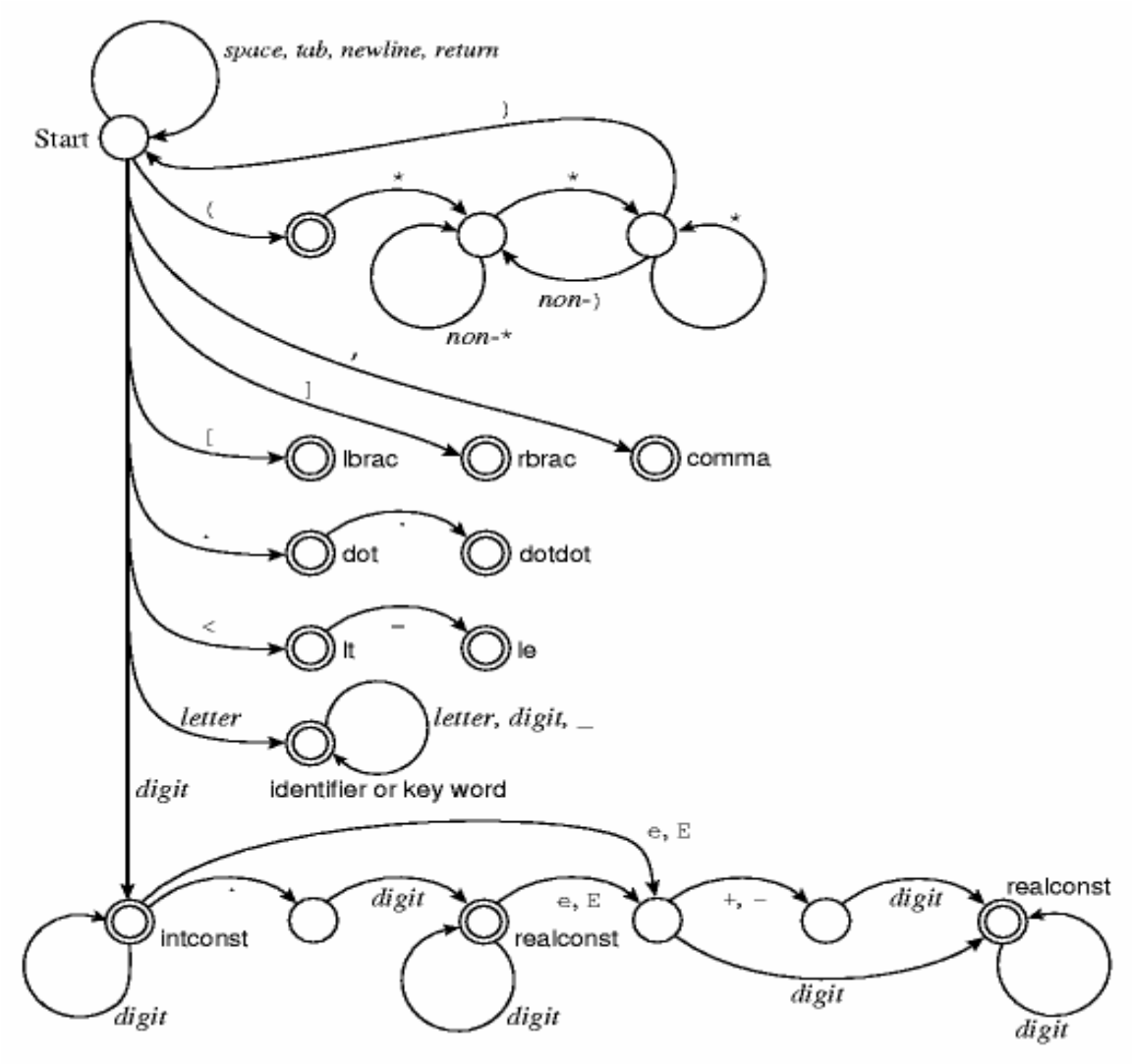
```
program gcd(input, output);
var i,j:integer;
begin
    read(i, j);
    while i<>j do
          if i> j then i:=i-j
          else j:= j-i;
    writeln(i)
end.
```

รูปที่ 4.6 โปรแกรมต้นทางในภาษาปาสคาล



รูปที่ 4.7 สายของโทเค็นจากการวิเคราะห์ศัพท์ของโปรแกรมต้นทางในรูปที่ 4.6

โปรแกรมคอมไพเลอร์ที่ทำหน้าที่วิเคราะห์ศัพท์ จึงมีชื่อเรียกว่า *สแกนเนอร์* (scanner) หรือ *เล็กซิคอลอนาไลเซอร์* (lexical analyzer) รูปแบบการทำงานจำลองได้ด้วยเครื่องจักรสมมุติที่เรียกว่า เครื่องจักรสถานะจำกัด (finite-state automata) ซึ่งเป็นโมเดลอย่างเป็นทางการ (formal model) ที่ใช้อธิบายการทำงานของสแกนเนอร์ ดังตัวอย่างในรูปที่ 4.8 ที่แสดงภาพบางส่วนของเครื่องจักรสถานะจำกัดที่ทำหน้าที่ตรวจจับคำศัพท์ประเภทต่างๆของโปรแกรมภาษาปาสคาลซึ่งถ้าจะแปลงการทำงานตามภาพเป็นสแกนเนอร์ที่ทำงานได้บนเครื่องคอมพิวเตอร์จริง ขั้นตอนการทำงานของโปรแกรมสแกนเนอร์จะเป็นดังในรูปที่ 4.9



รูปที่ 4.8 เครื่องจักรสถานะจำกัดแสดงการตรวจจับคำศัพท์ (บางส่วน) ของภาษาปาสคาล

```
we skip any initial white space (spaces, tabs, and newlines)
we read the next character
if it is a ( we look at the next character
      if that is a * we have a comment;
            we skip forward through the terminating *)
      otherwise we return a left parenthesis and reuse the look-ahead
if it is one of the one-character tokens ([ ] , ; = + - etc.)
      we return that token
if it is a . we look at the next character
      if that is a . we return ..
      otherwise we return . and reuse the look-ahead
if it is a < we look at the next character
      if that is a = we return <=
      otherwise we return < and reuse the look-ahead
etc.
```

รูปที่ 4.9 ขั้นตอนวิธีของสแกนเนอร์ที่ตรวจจับคำศัพท์(บางส่วน)ของภาษาปาสคาล

จากขั้นตอนวิธีในรูปที่ 4.9 ถ้าต้องการเขียนโปรแกรมสแกนเนอร์ด้วยภาษา Java จะมีโครงสร้างโปรแกรมดังรูปที่ 4.10

```
public Token nextToken( ) { // returns next token type and value
        Token t = new Token( );
        t.type = "Other";
        t.value = " ";
// First check for WhileSpace and bypass it
        while (isWhiteSpace (nextChar))  {
                nextChar = readChar( );
        }
//Then check for a comment and skip it.
// Comment in Pascal has a form (*...*)
        if  (nextChar = = '(' )  {
                if (nextChar = = '*')
                        t.type = "Comment";
                else
                        t.type = "LeftParenthesis";
                ...
        }
        ...
// Check for Identifier
        if (isLetter(nextChar))  {
                t.type = "Identifier";
                ...
        }
        ...
        return t;
}
```

รูปที่ 4.10 โครงสร้างโปรแกรมสแกนเนอร์ที่เขียนด้วยภาษา Java

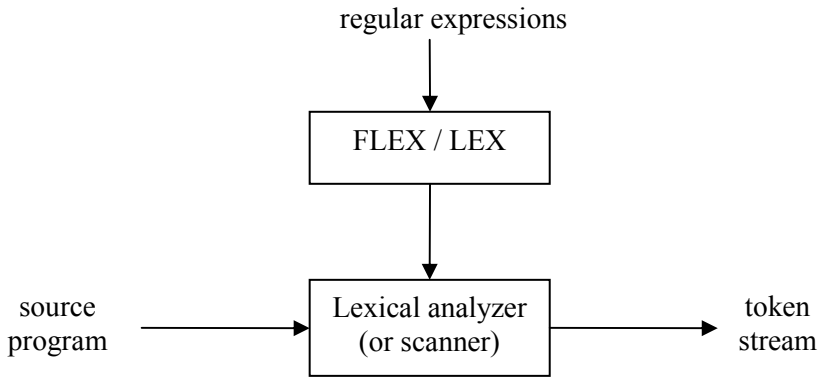
ในระหว่างการแยกคำออกเป็นโทเค็น สแกนเนอร์จะวิเคราะห์ประเภทของคำด้วยว่าคำนั้นๆ เป็นคำสงวน, ตัวแปร, ตัวกระทำการ, ตัวเลข หรือคำประเภทอื่นๆ ตามที่ระบุไว้ในไวยากรณ์คำศัพท์ (lexical syntax) ของภาษา ตัวอย่างเช่น รูปแบบ BNF ต่อไปนี้ ระบุไวยากรณ์คำศัพท์ที่ถูกต้องของการตั้งชื่อตัวแปรว่า จะต้องเริ่มต้นด้วยตัวอักษร แล้วอาจตามด้วยตัวอักษร หรือตัวเลข เป็นจำนวนเท่าใดก็ได้

```
Identifier   →  Letter | Identifier Letter | Identifier Digit
Letter       →  a | b| ... | z | A | B | ... | Z
Digit        →  0 | 1 | ... | 9
```

การระบุไวยากรณ์คำศัพท์ นอกจากใช้รูปแบบ BNF แล้วเรายังสามารถระบุไวยากรณ์ในรูปแบบอื่น ได้แก่ นิพจน์ปรกติ (regular expression) ที่มีความหมายเทียบเท่ากันดังนี้

```
[a-z A-Z][a-z A-Z 0-9]*
```

การระบุไวยากรณ์คำศัพท์ด้วยรูปแบบนิพจน์ปรกติ เป็นที่นิยมมากกว่าการระบุในรูปแบบ BNF เนื่องจากในปัจจุบันมีโปรแกรมช่วยสร้างคอมไพเลอร์ที่ทำหน้าที่สร้างโปรแกรมสแกนเนอร์ (หรือเล็กซิคอลอนาไลเซอร์) ให้เราโดยอัตโนมัติ เพียงแต่ระบุไวยากรณ์คำศัพท์ในรูปแบบนิพจน์ปรกติให้กับโปรแกรมช่วยงานนี้เท่านั้น (แสดงโครงสร้างได้ดังรูปที่ 4.11) โปรแกรมช่วยงานดังกล่าวคือโปรแกรม LEX เป็นชื่อย่อจากคำว่า LEXical analyzer generator ซึ่งหมายถึงโปรแกรมสร้างเล็กซิคอลอนาไลเซอร์ ผลลัพธ์ที่ได้จะเป็นโปรแกรมในรูปแบบภาษา C ปัจจุบันมีโปรแกรม JLEX ที่สร้างเล็กซิคอลอนาไลเซอร์เป็นภาษา Java โปรแกรม LEX ทำงานบนระบบปฏิบัติการยูนิกซ์ โปรแกรมประเภทเดียวกันที่ทำงานบนระบบปฏิบัติการวินโดส์คือ FLEX โปรแกรม LEX/JLEX/FLEX เมื่อทำงานเสร็จจะได้ผลลัพธ์เป็นโปรแกรมส่วนเล็กซิคอลอนาไลเซอร์ของคอมไพเลอร์ บางครั้งจึงเรียกโปรแกรมพวกนี้ว่า คอมไพเลอร์ที่สร้างคอมไพเลอร์ (compiler-compiler)



รูปที่ 4.11 โปรแกรมช่วยสร้างคอมไพเลอร์ส่วนเล็กซิคอลอนาไลเซอร์

ในขั้นตอนของการวิเคราะห์ศัพท์ โปรแกรมเล็กซิคอลอนาไลเซอร์นอกจากจะทำหน้าที่สร้างสายของโทเค็นแล้ว ยังมีการสร้างตารางสัญลักษณ์ (symbol table) เพื่อบันทึกชื่อตัวแปรและชื่อฟังก์ชันต่างๆ พร้อมทั้งชนิดของชื่อเหล่านั้น (เช่น ตัวแปรปรกติ, ตัวแปรอาร์เรย์, ฟังก์ชัน, ฟอร์มอลพารามิเตอร์) และชนิดของค่า (เช่น integer, real) รวมถึงข้อมูลอื่นๆ ที่จำเป็นต่อการตรวจสอบไวยากรณ์ของโปรแกรมต้นทาง ตารางสัญลักษณ์จะถูกสร้างตั้งแต่ขั้นตอนแรกของการคอมไพล์โปรแกรม และจะถูกใช้งานตลอดการทำงานของคอมไพเลอร์

<u>ขั้นตอนที่ 2 การวิเคราะห์ไวยากรณ์</u> (Syntactic analysis or parsing)

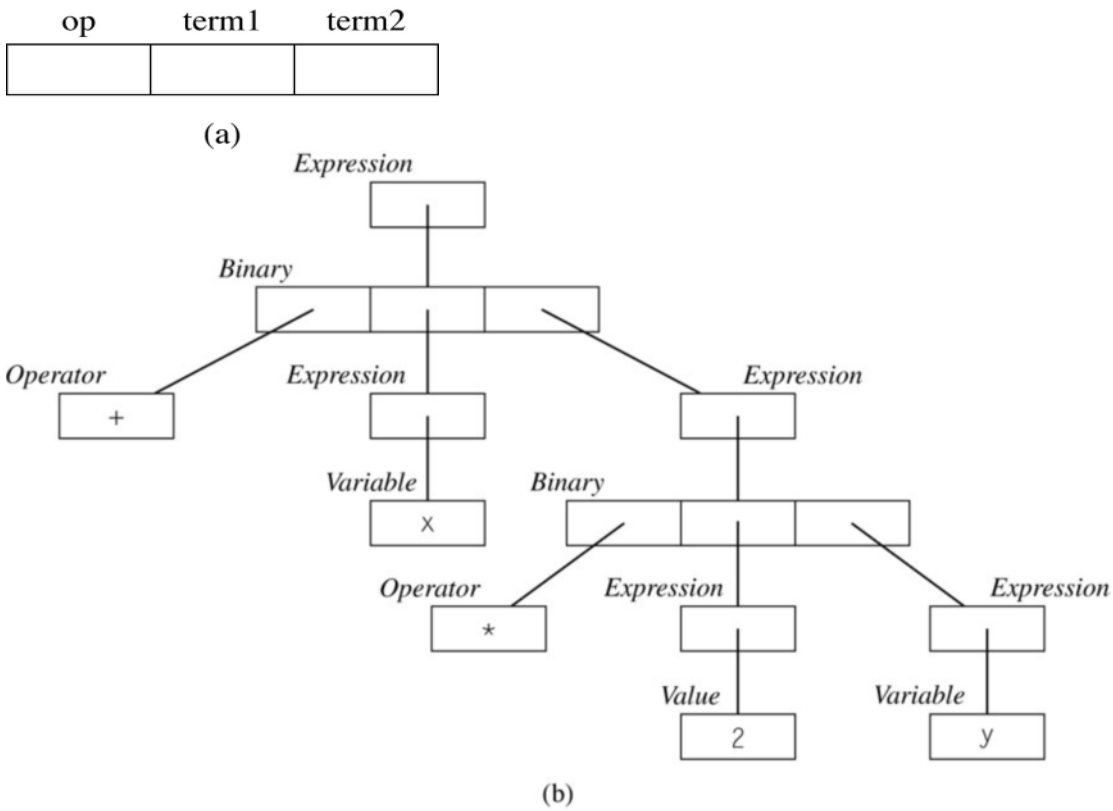
โปรแกรมคอมไพเลอร์ที่ทำหน้าที่ในส่วนนี้มีชื่อเรียกเฉพาะว่า โปรแกรมซินแท็กติกอนาไลเซอร์ (syntactic analyzer) หรือนิยมเรียกในอีกชื่อหนึ่งว่า พาสเซอร์ (parser) โปรแกรมนี้ทำหน้าที่ตรวจสอบว่าสายของโทเค็นนั้น ประกอบกันขึ้นเป็นนิพจน์ที่ถูกต้อง, เป็นข้อความสั่งที่ถูกต้อง, และสุดท้ายเป็นรูปแบบโปรแกรมที่ถูกต้องตามข้อกำหนดของภาษาหรือไม่ การตรวจสอบความถูกต้องจะใช้วิธีการเทียบเคียงกับ

ไวยากรณ์ของภาษาที่เขียนอยู่ในรูปแบบ BNF ดังตัวอย่างไวยากรณ์ต่อไปนี้เป็นการกำหนดรูปแบบการเขียนประโยคข้อความสั่งกำหนดค่าให้กับตัวแปร

```
Assignment  →  Identifier = Expression ;
Expression  →  Term | Expression + Term | | Expression - Term
Term        →  Factor| Term*Factor| Term/Factor
Factor      →  Identifier | Literal | (Expression)
```

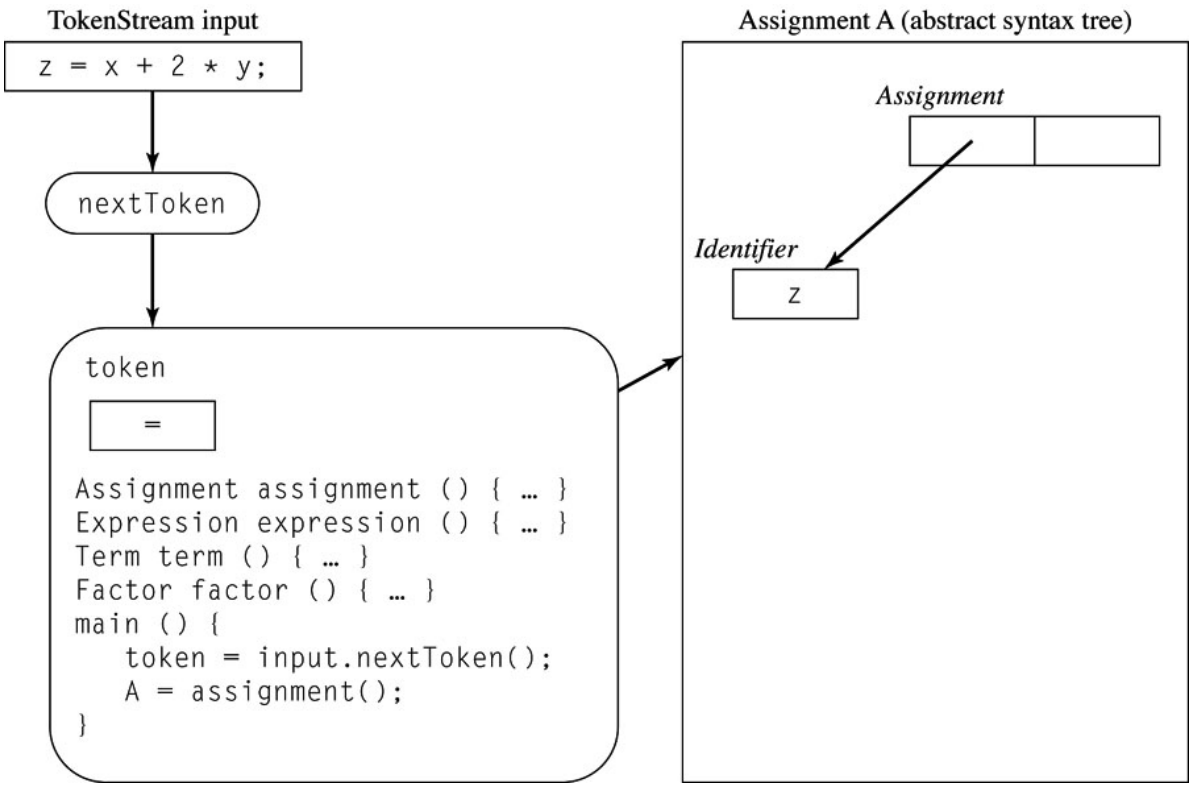
การกำหนดรูปแบบไวยากรณ์ในระดับประโยคข้อความหรือในระดับโปรแกรมทั้งโปรแกรมนี้ ไม่สามารถใช้ไวยากรณ์ปรกติ (regular grammar) ที่เขียนอยู่ในรูปแบบนิพจน์ปรกติเพราะขีดความสามารถต่ำเช่น ไม่สามารถระบุได้ว่านิพจน์ที่มีวงเล็บ จะต้องมีจำนวนวงเล็บเปิดเท่ากับจำนวนวงเล็บปิด จึงต้องใช้ไวยากรณ์ในระดับที่สูงขึ้นเรียกว่า ไวยากรณ์ไม่พึ่งบริบท (context-free grammar) ที่นิยมเขียนอยู่ในรูปแบบบีเอ็นเอฟดังตัวอย่างไวยากรณ์ `Assignment` ข้างต้น

การตรวจสอบความถูกต้องในลักษณะการวิเคราะห์ไวยากรณ์ของประโยคข้อความสั่ง จะใช้วิธีการสร้างพาสทรี ถ้าสามารถสร้างพาสทรีที่สมบูรณ์ครบถ้วนได้แสดงว่าโปรแกรมต้นทางเขียนได้ถูกต้องตามไวยากรณ์ของภาษา และพาสทรีที่สร้างขึ้นจะถูกส่งต่อไปให้ขั้นตอนต่อไปของคอมไพเลอร์ รูปที่ 4.12 แสดงพาสทรีในการตรวจสอบนิพจน์ *x+2*y* พาสทรีที่แสดงในรูปเป็นพาสทรีที่มีรูปแบบกะทัดรัดขึ้นเรียกว่า abstract syntax tree ที่มีการเปลี่ยนนิพจน์ให้อยู่ในรูปแบบพรีฟิกซ์



รูปที่ 4.12 โครงสร้างพาสทรีเพื่อตรวจสอบนิพจน์ *x+2*y*

รูปที่ 4.13 แสดงการทำงานของโปรแกรมพาสเซอร์ที่รับสายโทเค็น แล้ววิเคราะห์ไวยากรณ์ด้วยโครงสร้างทรี ตัวอย่างฟังก์ชัน assignment เพื่อตรวจสอบข้อความสั่งกำหนดค่าให้กับตัวแปรแสดงได้ (ด้วยรูปแบบภาษา Java) ดังรูปที่ 4.14

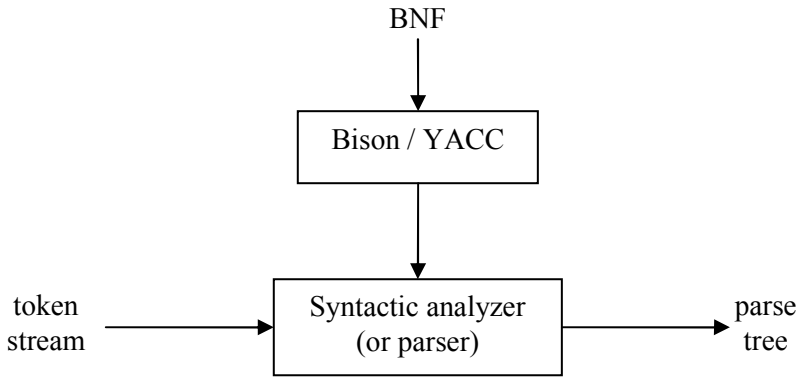


รูปที่ 4.13 โครงสร้างของโปรแกรมพาสเซอร์ที่วิเคราะห์ข้อความสั่งกำหนดค่า

```
private Assignment assignment ( ) {
        Assignment a = new Assignment ( );
        if (token.type.equals("Identifier")) {
                a.target = new Variable ( );
                a.target.id = token.value ;
                token = input.nextToken ( );
                match ("=");
                a.source = expression ( );
                match (";");
        }
        else  SyntaxError ("Identifier");
        return a;
}
```

รูปที่ 4.14 ฟังก์ชัน assignment ในโปรแกรมพาสเซอร์

โปรแกรมพาสเซอร์สามารถถูกสร้างขึ้นด้วยการใช้โปรแกรมช่วย ที่มีชื่อเรียกว่าโปรแกรม YACC (Yet Another Compiler – Compiler) ซึ่งจะสร้างพาสเซอร์ในรูปแบบของโปรแกรมภาษา C โดย YACC จะทำงานบนระบบปฏิบัติการยูนิกซ์ ในกรณีของวินโดส์ จะมีโปรแกรมชื่อ Bison ที่ทำหน้าที่สร้างพาสเซอร์ให้เช่นเดียวกัน โดยผู้ใช้จะต้องระบุไวยากรณ์ BNF ให้กับโปรแกรม (ตามโครงสร้างในรูปที่ 4.15)



รูปที่ 4.15 โปรแกรมช่วยสร้างคอมไพเลอร์ส่วนพาสเซอร์

<u>ขั้นตอนที่ 3 การวิเคราะห์ความหมาย</u> (Semantic analysis)

ขั้นตอนการวิเคราะห์ความหมายเป็นการตรวจสอบความถูกต้องของโปรแกรมต้นทาง การตรวจสอบนี้กระทำในระดับที่ซับซ้อนขึ้นกว่าการตรวจสอบไวยากรณ์ของข้อความสั่งต่างๆ เช่น ตรวจสอบว่าตัวแปรที่ใช้ในข้อความสั่งมีการประกาศไว้ก่อนที่จะใช้งาน, ตรวจสอบชนิดของข้อมูลว่าสามารถทำงานด้วยกันได้ (เช่น ไม่นำตัวแปรสตริงไปคูณกับตัวแปร integer), ตรวจสอบพารามิเตอร์ว่า formal parameters และ actual parameters มีชนิดที่ตรงกัน

<u>ขั้นตอนที่ 4 การสร้างรหัสระหว่างกลาง</u> (Intermediate code generation)

การทำงานของคอมไพเลอร์ในขั้นตอนที่ 1-3 เป็นช่วงการตรวจสอบความถูกต้องของโปรแกรมต้นทาง ขั้นตอนที่ 4-6 จะเป็นช่วงของการแปลงไปสู่โปรแกรมภาษาเครื่อง แต่การแปลงจะไม่แปลงเป็นภาษาเครื่องในทันทีแต่จะแปลงไปสู่ภาษาที่เป็นกึ่งกลางระหว่างภาษาระดับสูงและภาษาเครื่อง ภาษากึ่งกลางนี้เรียกว่า รหัสระหว่างกลาง (intermediate code) ทั้งนี้เพื่อประโยชน์ในการปรับปรุงคำสั่งให้สามารถประมวลผลได้รวดเร็วขึ้น มีประสิทธิภาพมากขึ้น ทั้งนี้เพราะการปรับปรุงใดๆ กับคำสั่งในภาษาเครื่องทำได้ไม่สะดวก จึงต้องสร้างรหัสคำสั่งในระดับที่สูงกว่าภาษาเครื่อง รหัสระหว่างกลางที่นิยมใช้จะเรียกว่า three-address code ที่อยู่ในรูปแบบ

```
result = onething operator anotherthing
```

ตัวอย่างเช่น

```
T = rate * time
```

<u>ขั้นตอนที่ 5 การเพิ่มประสิทธิภาพคำสั่ง</u> (Code optimization)

เมื่อโปรแกรมต้นทางถูกแปลงให้อยู่ในรูปแบบรหัสระหว่างกลางแล้ว โปรแกรมคอมไพเลอร์ส่วนที่เรียกว่า code optimizer จะทำหน้าที่ปรับปรุงลำดับการสั่งงานในคำสั่งให้ดีขึ้น ตัวอย่างเช่น จากคำสั่งกำหนดค่าในโปรแกรมต้นทาง

```
x = a * y + z;
```

เมื่อแปลงเป็นรหัสระหว่างกลางในลักษณะ three-address code จะได้รูปแบบ

$$T_1 = a*y$$
$$T_2 = T_1+z$$
$$x = T_2$$

โดย T_1และ T_2 เป็นตัวแปรชั่วคราวที่ถูกสร้างขึ้นโดยคอมไพเลอร์เพื่อใช้เก็บผลลัพธ์ระหว่างการคำนวณนิพจน์ การเพิ่มประสิทธิภาพคำสั่งสามารถทำได้โดยการยุบรวมคำสั่งที่สองและคำสั่งที่สาม ได้เป็น

$$T_1 = a*y$$
$$x = T_1+z$$

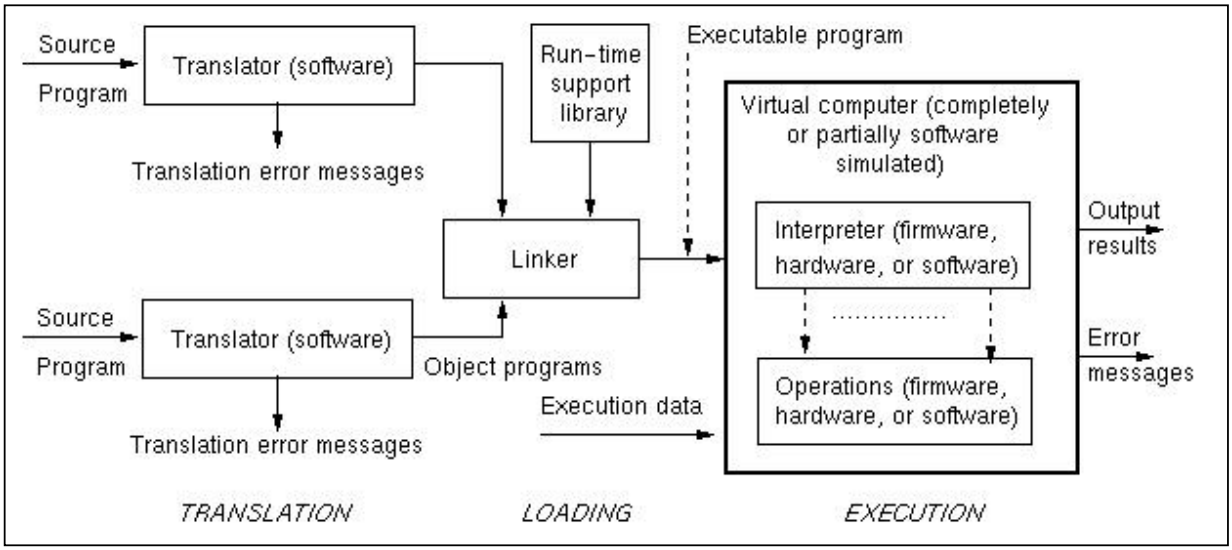
<u>ขั้นตอนที่ 6 การสร้างรหัสเป้าหมาย</u> (Object code generation)

รหัสระหว่างกลางที่ได้รับการปรับปรุงประสิทธิภาพแล้ว จะถูกแปลงเป็นรหัสเป้าหมาย (object code) ซึ่งจะขึ้นกับเครื่องคอมพิวเตอร์ที่ใช้ จากตัวอย่าง three-address code ในขั้นตอนก่อนหน้านี้สามารถแปลงเป็นรหัสเป้าหมายตามรูปแบบของเครื่อง IBM / 370 ได้เป็น

```
LE    A, 4    // A in floating-point register 4
ME    4, Y    // Multiply by Y
AE    4, Z    // Add Z
STE   4, X    // Store in X
```

รหัสเป้าหมายนี้อยู่ในรูปแบบภาษาแอสเซมบลีที่จะถูกแปลงเป็นภาษาเครื่องได้ต่อไป

จากตัวอย่างการทำงานของคอมไพเลอร์ที่ในขั้นตอนสุดท้ายจะได้รหัสคำสั่งภาษาเครื่อง (จากตัวอย่างเป็นเครื่อง IBM/370) ทำให้รหัสคำสั่งที่ได้ทำงานได้กับเครื่องคอมพิวเตอร์เฉพาะตระกูลนั้นๆ เท่านั้น เมื่อเปลี่ยนเครื่องเป็นรุ่นอื่นแบบอื่นรหัสคำสั่งเดิมที่มีอยู่จะทำงานไม่ได้ การพัฒนาคอมไพเลอร์ในระยะหลังจึงเปลี่ยนวิธีการทำงาน จากการแปลให้ได้รหัสเป้าหมายตามรูปแบบของเครื่องคอมพิวเตอร์ (เช่น IBM/370) เป็นการแปลงไปสู่รหัสเป้าหมายของเครื่องคอมพิวเตอร์เสมือน (ดังแผนภาพในรูปที่ 4.16) ซึ่งเป็นการจำลองการทำงานของเครื่องคอมพิวเตอร์ โดยใช้ซอฟต์แวร์แทนที่จะใช้อุปกรณ์ฮาร์ดแวร์จริง



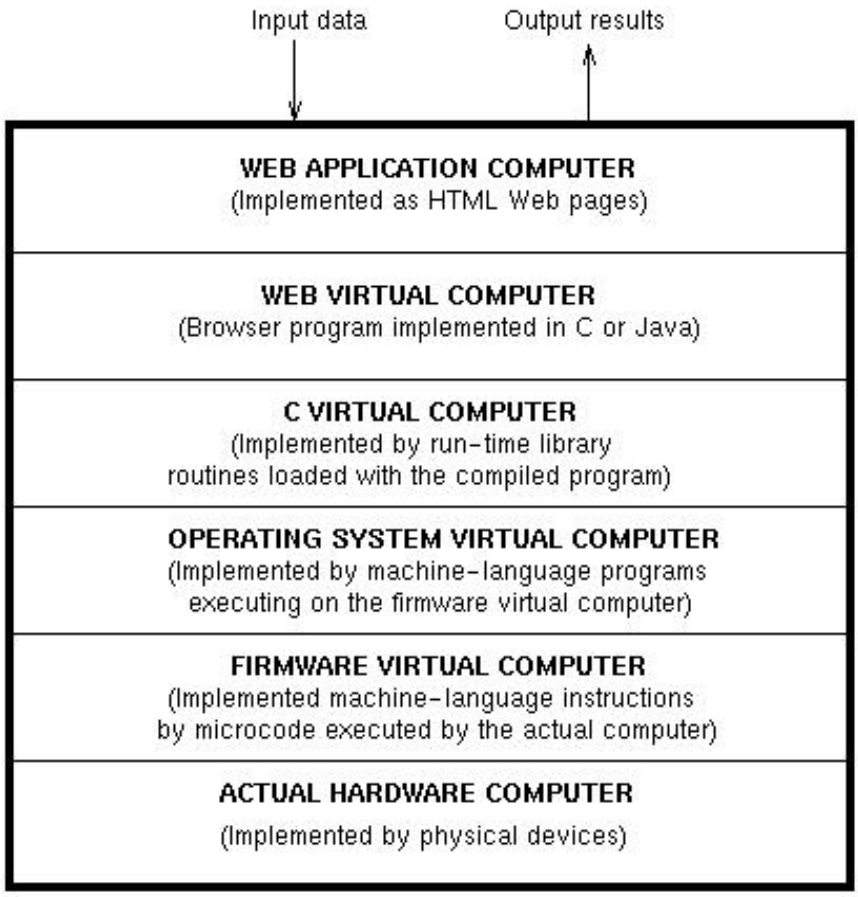
รูปที่ 4.16 ขั้นตอนการทำงานของคอมไพเลอร์ที่สร้างรหัสเป้าหมายบนเครื่องคอมพิวเตอร์เสมือน

4.3 เครื่องจักรเสมือน

(Virtual machine)

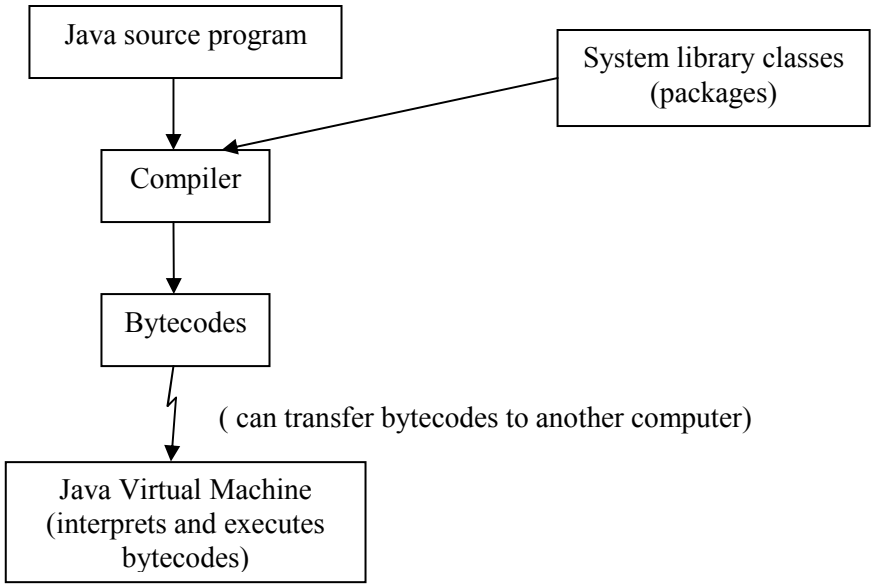
คำว่าเครื่องจักรเสมือน หรือในบางครั้งอาจเรียกว่า เครื่องจักรนามธรรม (abstract machines) จะหมายถึงการจำลองการทำงานของเครื่องคอมพิวเตอร์จริงที่เป็นฮาร์ดแวร์ด้วยซอฟต์แวร์ แนวคิดเกี่ยวกับเครื่องจักรเสมือนเป็นพื้นฐานสำคัญในงานการโปรแกรมเพื่อสั่งงานคอมพิวเตอร์ ตัวอย่างเช่น การพัฒนาโปรแกรมลงทะเบียนเรียนผ่านเว็บ ส่วนหน้าจอที่ติดต่อกับผู้ใช้จะเป็นเว็บเพจที่เขียนด้วยภาษา HTML โปรแกรม HTML นี้ จะต้องรันบนโปรแกรมเว็บบราวเซอร์ เช่น Internet Explorer โปรแกรมเว็บบราวเซอร์จึงทำหน้าที่เป็น เครื่องจักรเสมือนให้โปรแกรม HTML ทำงานได้

ถ้าโปรแกรมเว็บบราวเซอร์เขียนขึ้นด้วยภาษา C โปรแกรมนี้จะทำงานได้ก็ต้องอาศัยโปรแกรมใน C run-time library ซึ่งทำหน้าที่เป็นเครื่องจักรเสมือนประมวลผลโปรแกรมภาษา C และ C run-time library จะต้องอาศัยระบบปฏิบัติการ หรือ OS ทำหน้าที่เป็นเครื่องจักรเสมือนประมวลผลให้ โดย OS จะต้องอาศัยไมโครโค้ดทำหน้าที่เป็นเครื่องจักรเสมือน และสุดท้ายไมโครโค้ดจะทำงานบนเครื่องคอมพิวเตอร์จริงที่ประกอบขึ้นจากวงจรอิเล็กทรอนิกส์ต่างๆ ลำดับชั้นของเครื่องจักรเสมือนนี้แสดงเป็นแผนภาพได้ดังรูปที่ 4.17



รูปที่ 4.17 ลำดับชั้นของเครื่องจักรเสมือนในการทำงานกับโปรแกรมเว็บ

ผู้พัฒนาภาษา Java ได้ใช้แนวคิดในเรื่องเครื่องจักรเสมือน มาช่วยเพิ่มขีดความสามารถของภาษา Java ให้ประมวลผลบนเครื่องคอมพิวเตอร์ที่มีระบบแตกต่างกันได้ (เรียกคุณสมบัตินี้ว่า portability) โดยกลไกสำคัญที่ช่วยเพิ่มขีดความสามารถนี้คือ ซอฟต์แวร์ที่เรียกว่า Java Virtual Machine (JVM) โครงสร้างการประมวลผลโปรแกรมในภาษา Java แสดงได้ดังรูปที่ 4.18



รูปที่ 4.18 ขั้นตอนการประมวลผลโปรแกรมภาษา Java

จากโครงสร้างการประมวลผลโปรแกรมภาษา Java จะเห็นได้ว่าคอมไพเลอร์ทำหน้าที่แปลโปรแกรมจากรูปแบบภาษา Java ให้เป็นรูปแบบรหัสระหว่างกลาง เรียกว่า ไบต์โค้ด (bytecodes) ตัวอย่างรหัสไบต์โค้ดแสดงได้ดังรูปที่ 4.19 (คอมไพเลอร์ที่ใช้ช่วยในการคอมไพล์โปรแกรมให้เป็นไบต์โค้ดจะอยู่ในชุดซอฟต์แวร์ที่เรียกว่า Java Development Kit หรือ JDK สามารถดาวน์โหลดได้จากบริษัท Sun Microsystems)

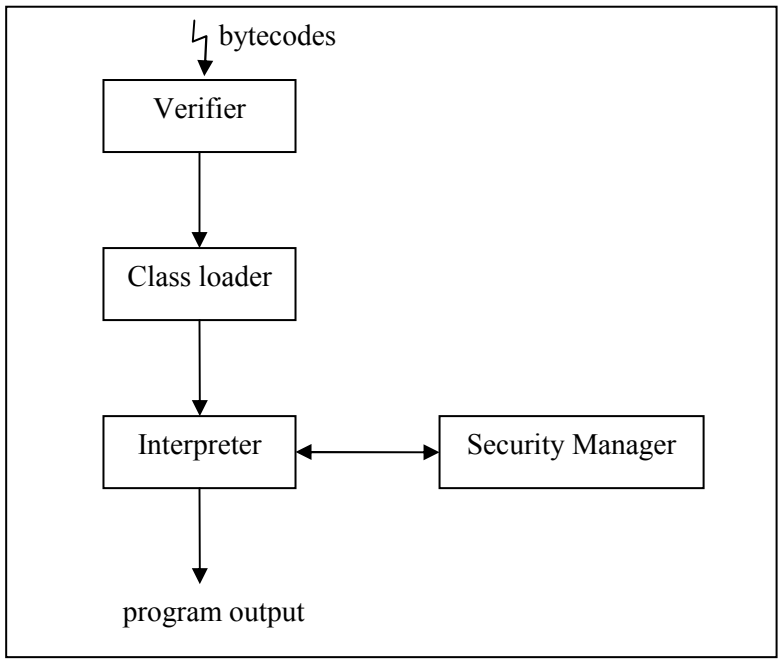
public class Hello
{
public static void main (String[] args)
{
System.out.println(“Hello,World!”);
}
}
	Compiled from Hello.java
	class Hello extends java.lang.Object
	{
	Hello( );
	public static void main (java.lang. String[]);
	}
	Method Hello( )
	0 aload_o
	1 invokespecial #6> Method java.lang.Object( ) <
	4 return
	Method void main(java.lang. String[])
	0 getstatic # 7 > Field java.io.PrintStream out <
	3 ldc # 1 > String “Hello,World!”<
	5 invoke virtual #8 > Method
	void println (java.lang. String[]) <
	8 return

(a) source program *Hello.java* (b) bytecodes *Hello.class*

รูปที่ 4.19 ตัวอย่างโปรแกรมในรูปแบบภาษา Java และรูปแบบ ไบต์โค้ด

คุณสมบัติการทำงานได้บนหลายระบบ (portability) ของภาษา Java เกิดจากการที่ JVM รับรูปแบบไบต์โค้ด แล้วแปลง(ด้วยอินเตอร์พรีตเตอร์ หรือคอมไพเลอร์) เป็นรหัสภาษาระดับต่ำบนเครื่องแต่ละระบบเช่น WINDOWS 98, WINDOWS XP, WINDOWS NT, UNIX, LINUX, Sun Solaris, Macintosh OS เป็นต้น ดังนั้นก่อนที่เครื่องคอมพิวเตอร์ใดจะคอมไพล์และรันโปรแกรม Java ได้ จะต้องมี JVM ติดตั้งบนเครื่องนั้นก่อน และ JVM จะมีหลายเวอร์ชันสำหรับระบบปฏิบัติการแต่ละประเภท เช่น JVM สำหรับ WINDOWS XP, JVM สำหรับ LINUX

การรันโปรแกรม Java นอกจากจะต้องมี JVM แล้ว ยังต้องมีโปรแกรมต่างๆ ที่ช่วยตรวจสอบความถูกต้องของการเรียกใช้คลาส โปรแกรมที่ทำหน้าที่เป็น interpreter และโปรแกรมช่วยงานต่างๆ โปรแกรมเหล่านี้รวมเรียกว่า Java Runtime Environment (JRE) ซึ่งแสดงโครงสร้างได้ดังรูปที่ 4.20



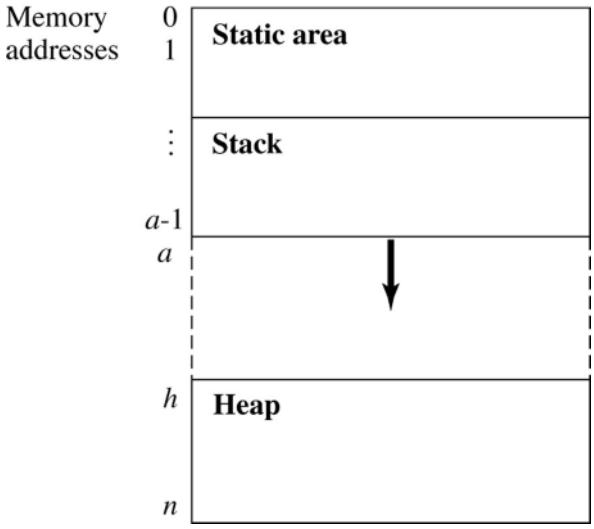
รูปที่ 4.20 โครงสร้าง Java Runtime Environment (JRE)

4.4 โครงสร้างหน่วยความจำขณะประมวลผล

(Run-time memory structure)

การประมวลผลโปรแกรมจำเป็นต้องใช้หน่วยความจำหลัก เป็นส่วนประกอบสำคัญในการเก็บ program source code , เก็บค่าของตัวแปรต่างๆ ขณะที่ประมวลผลโปรแกรม ตลอดจนถึงการเก็บข้อมูลต่างๆ ที่ต้องใช้ประกอบการติดตามการทำงานของโปรแกรม ดังนั้นการทำความเข้าใจเกี่ยวกับโครงสร้างหน่วยความจำและการจัดการหน่วยความจำ จึงมีความจำเป็นต่อการศึกษาเกี่ยวกับแนวคิดของการทำโปรแกรมและยังเป็นการช่วยให้เกิดความเข้าใจที่ลึกซึ้งขึ้นเกี่ยวกับวิธีการทำงานของโปรแกรม

โครงสร้างของหน่วยความจำส่วนที่เกี่ยวข้องกับการประมวลผลโปรแกรม จะประกอบด้วยหน่วยความจำสามส่วน เรียกว่า บริเวณคงตัว (static area), สแตกขณะดำเนินงาน (rut-time stack) และฮีพ (heap) ทั้งสามส่วนนี้เป็นเนื้อที่หน่วยความจำที่ใช้เก็บค่าของตัวแปรและข้อมูลต่างๆ ที่เกี่ยวข้อง โครงสร้างแสดงเป็นแผนภาพได้ดังรูปที่ 4.21



รูปที่ 4.21 โครงสร้างหน่วยความจำขณะมีการประมวลผลโปรแกรม

พื้นที่หน่วยความจำบริเวณคงตัว ใช้เก็บค่าของตัวแปรที่รู้ค่าล่วงหน้าและค่านั้นคงอยู่ตลอดการทำงานของโปรแกรม พื้นที่ส่วนสแตกขณะดำเนินงาน (หรือเรียกสั้นๆ ว่า สแตก ซึ่งในที่นี้จะหมายถึงสแตกของระบบคอมพิวเตอร์ไม่ใช่โครงสร้างข้อมูลสแตกที่ใช้ในโปรแกรม) เป็นพื้นที่หลักที่มีการใช้งานขณะประมวลผลโปรแกรม โดยจะใช้เก็บค่าของตัวแปรต่างๆ ที่สามารถเปลี่ยนค่าได้ตลอดการทำงานของโปรแกรม, เก็บค่าพารามิเตอร์เมื่อมีการเรียกใช้โปรแกรมย่อย และเก็บการเชื่อมโยงการทำงานเมื่อโปรแกรมย่อย เรียกใช้โปรแกรมย่อยอื่นต่อเนื่องไปอีกหลายทอด และสุดท้ายพื้นที่ส่วนฮีพใช้เก็บค่าของข้อมูลที่เกิดขึ้นขณะรันคำสั่งในโปรแกรม

การจัดการหน่วยความจำหรือ การจัดสรรพื้นที่หน่วยความจำขณะโปรแกรมถูกประมวลผล จะสอดคล้องกับการแบ่งประเภทพื้นที่หน่วยความจำ หน่วยความจำบริเวณคงตัวจะถูกใช้เมื่อมีการประกาศตัวแปรประเภทคงตัว (static) ดังตัวอย่างโปรแกรม C/C++ ต่อไปนี้

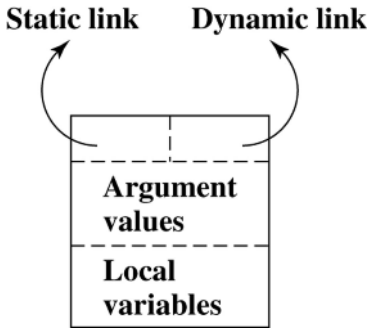
```
void DoSomeStuff (int data)
{
        static int invocations = 0;
        int x, y, z;
        // do whatever the function is supposed to do
        invocations ++;
}
```

ตัวแปรทั้งหมดในตัวอย่าง (*data, invocations, x, y, z*) จะได้รับการจัดสรรพื้นที่หน่วยความจำเมื่อฟังก์ชัน *DoSomeStuff* ถูกเรียกใช้ (เช่น ถูกเรียกจากฟังก์ชัน main) ตัวแปร *data, x, y, z* จะอยู่ในพื้นที่หน่วยความจำส่วนสแตก และพื้นที่ที่ได้รับจัดสรรสามารถถูกเรียกคืน (เพื่อนำพื้นที่ไปใช้เก็บค่าตัวแปรอื่นๆ) โดยระบบปฏิบัติการเมื่อฟังก์ชันจบการทำงาน แต่ตัวแปร *invocations* จะได้รับจัดสรรพื้นที่ในส่วนบริเวณคงตัว และจะใช้พื้นที่นั้นไปตลอดการทำงานของโปรแกรม ถึงแม้ฟังก์ชัน *DoSomeStuff* ทำงานเสร็จ พื้นที่ของตัวแปร *invocations* ก็จะยังคงใช้เก็บค่าอยู่ จนกระทั่งโปรแกรมจบการทำงานพื้นที่นี้จึงจะถูกเรียกคืน

ในกรณีที่โปรแกรมมีการเรียกใช้โปรแกรมย่อยหลายโปรแกรมดังตัวอย่างต่อไปนี้ที่ฟังก์ชัน main เรียกใช้ฟังก์ชัน A และฟังก์ชัน A เรียกใช้ฟังก์ชัน B

```
package K {
        int   h, i;
        void A (int x, int y)  {
                boolean i, j;
                B (h);
                ...
        }
        void B(int w)  {
                int j, k;
                i = 2*w;
                w = w+1;
                ...
        }
void main ( )  {
        int a, b;
        h=5; a=3; b=2;
        A=(a,b);
        ...
}
```

ทุกครั้งที่โปรแกรมมีการเรียกใช้โปรแกรมย่อย (รวมถึงโปรแกรมย่อยที่เป็นฟังก์ชัน main) พื้นที่หน่วยความจำส่วนสแตกจะได้รับการจัดสรรให้กับแต่ละโปรแกรมย่อย และแต่ละพื้นที่จะมีชื่อเรียกว่า สแตกเฟรม (stack frame) หรือบางครั้งเรียกว่า เรคคอร์ดการกระตุ้น (activation recode) ซึ่งมีโครงสร้างของเฟรมดังรูปที่ 4.22

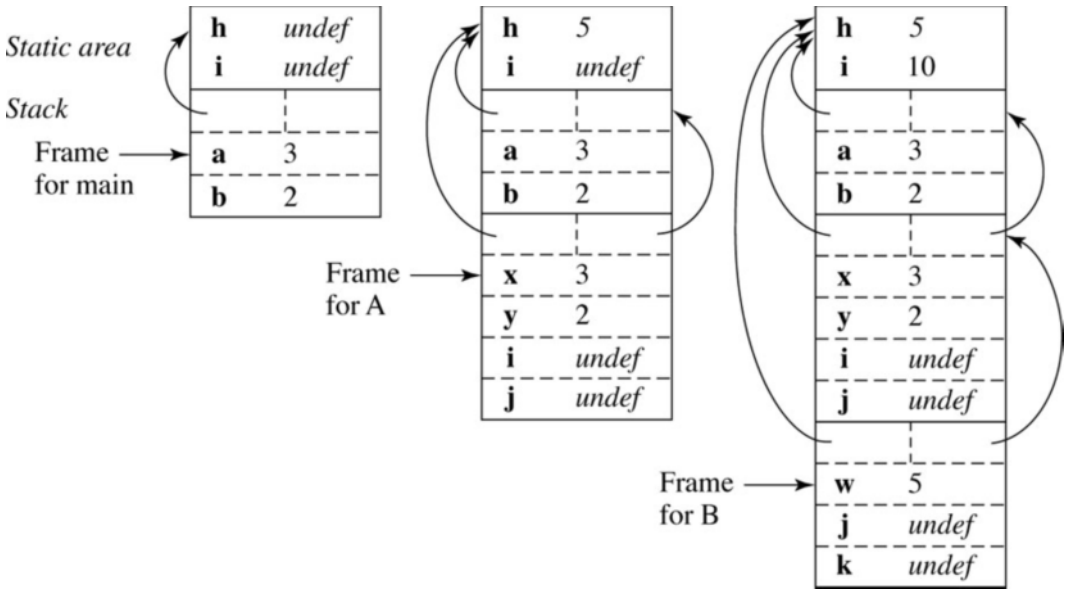


รูปที่ 4.22 โครงสร้างของสแตกเฟรม

เมื่อโปรแกรมตามตัวอย่างข้างต้นเริ่มทำงาน ตัวแปร h และ i ที่ประกาศไว้ต้นโปรแกรม และประกาศไว้เหนือขอบเขตของทุกฟังก์ชัน (ขอบเขตของฟังก์ชันจะอยู่ภายในวงเล็บปีกกา เช่น void main() {…ขอบเขตของฟังก์ชัน main ...}) การเก็บค่าของ h และ i จะถูกจัดสรรเนื้อที่ไว้ที่บริเวณคงตัว โดยค่าของตัวแปร h และ i จะถูกกำหนดให้เป็น undef เนื่องจากตัวแปรทั้งสองยังไม่ได้ถูกกำหนดค่าเริ่มต้น ต่อจากนั้นฟังก์ชัน main จะเป็นฟังก์ชันแรกที่เริ่มทำงาน ในฟังก์ชัน main มีการใช้ตัวแปร a, b และเริ่มมีการกำหนดค่าให้กับตัวแปร h ด้วยคำสั่ง *h=5;* ดังนั้นในโครงสร้างของหน่วยความจำ (ดังรูปที่ 4.23) จะมีการจัดสรรเนื้อที่ส่วนสแตกให้กับฟังก์ชัน main เพื่อใช้เก็บค่าของตัวแปร a, b และเก็บตัวชี้เรียกว่า สายเชื่อมโยงคงตัว (static link) เพื่อเชื่อมโยงไปที่ตัวแปร h ในบริเวณคงตัว และสามารถเปลี่ยนค่าของตัวแปร h ได้

ในตัวอย่างข้างต้น ตัวแปร h และ i จะเรียกว่า *ตัวแปรส่วนกลาง* (global variables) ในขณะที่ตัวแปร a และ b จะเรียกว่า *ตัวแปรเฉพาะที่* (local variables) ตัวแปรทั้งสองประเภทจะมีข้อแตกต่างกันในสามประเด็นหลักคือ

- ตำแหน่งที่เก็บตัวแปรในหน่วยความจำ (ตัวแปรส่วนกลางอยู่ที่หน่วยความจำบริเวณคงตัว แต่ตัวแปรเฉพาะที่ใช้หน่วยความจำสแตก),
- ขอบเขต (scope) ของตัวแปร (ตัวแปรส่วนกลางมีขอบเขตการใช้งานครอบคลุมทั้งโปรแกรม ทุกฟังก์ชันสามารถเรียกใช้ได้ แต่ตัวแปรเฉพาะที่มีขอบเขตการใช้งานเฉพาะภายในฟังก์ชันที่ประกาศใช้ตัวแปรนั้นเท่านั้น),
- ช่วงอายุ (life-time) ของตัวแปร (ตัวแปรส่วนกลางมีช่วงอายุยาวตลอดการทำงานของโปรแกรม แต่ตัวแปรเฉพาะที่มีช่วงอายุเพียงขณะที่ฟังก์ชันที่ประกาศใช้ตัวแปรนั้นยังทำงานอยู่เท่านั้น เมื่อฟังก์ชันทำงานจบ ตัวแปรเฉพาะที่จะหมดอายุตามไปด้วย)



รูปที่ 4.23 โครงสร้างหน่วยความจำขณะฟังก์ชัน main ทำงานและขณะมีการเรียกใช้ฟังก์ชัน A และ B

เมื่อฟังก์ชัน main เรียกใช้ฟังก์ชัน A จะมีการสร้างสแตกเฟรมเพื่อใช้เก็บค่าตัวแปรเฉพาะที่ของฟังก์ชัน A ซึ่งประกอบด้วยตัวแปร i, j รวมถึงตัวแปร x, y ที่เป็นพารามิเตอร์ (เรียกว่า formal parameter) รับค่า 3, 2 (เรียกว่า actual parameter) มาจากฟังก์ชัน main ในสแตกเฟรมของฟังก์ชัน A ยังมีตัวชี้ที่เรียกว่าสายเชื่อมโยงพลวัต (dynamic link) เพื่อเชื่อมโยงกลับไปที่สแตกเฟรมของฟังก์ชัน main ที่เป็นผู้เรียกใช้ (caller) ฟังก์ชัน A

เมื่อฟังก์ชัน A เริ่มทำงานมีการเรียกใช้ ฟังก์ชัน B พร้อมทั้งส่งค่า h (=5) เป็นพารามิเตอร์ไปให้ฟังก์ชัน B จะมีการสร้างสแตกเฟรมเพิ่มอีกหนึ่งเฟรมสำหรับเก็บค่าตัวแปรต่างๆ ของฟังก์ชัน B และสายเชื่อมโยงพลวัต จะชี้กลับไปที่ฟังก์ชัน A ที่เป็นผู้เรียก ส่วนสายเชื่อมโยงคงตัวจะชี้ไปที่บริเวณคงตัวที่เก็บค่าตัวแปร h และ i เมื่อฟังก์ชัน B ทำงานเสร็จ สแตกเฟรมของ B จะถูกยกเลิก หรือเรียกว่าถูกพ็อพ (pop) และเมื่อฟังก์ชัน A ทำงานเสร็จแล้ว สแตกเฟรมของ A ก็จะถูกพ็อพเช่นเดียวกัน สุดท้ายเมื่อฟังก์ชัน main ทำงานเสร็จ สแตกเฟรมของ main ก็จะถูกพ็อพ เมื่อโปรแกรมจบการทำงานจะยกเลิกเนื้อที่สำหรับตัวแปร h และ i ในบริเวณคงตัว

```
void  A (int x, int y)  {
        boolean  i, j ;
        B(h) ;
        ...
}
```

```
void  B (int w)  {
        int  j, k ;
        i = 2*w ;
        w = w+1 ;
        ...
}
```

จากโปรแกรมตัวอย่างมีการประกาศตัวแปร h และ i เป็นตัวแปรส่วนกลาง เป็นชนิด int แต่ในฟังก์ชัน A มีการใช้ชื่อตัวแปร i อีกครั้ง เป็นชนิด boolean ชื่อตัวแปร i นี้ใช้ซ้ำกันได้ เพราะตำแหน่งใน

หน่วยความจำ ที่ใช้เก็บค่าของ i อยู่ในตำแหน่งแตกต่างกัน และเมื่อมีการอ้างอิงถึงชื่อ i (ดังคำสั่งในฟังก์ชัน B) ถ้าตรวจสอบแล้วพบว่า i ไม่ใช่ตัวแปรเฉพาะที่ (ตัวแปรที่ประกาศอยู่ภายในฟังก์ชัน B)จะมีการเชื่อมโยงไปยังบริเวณคงตัวเพื่อนำค่า i ที่เป็นตัวแปรส่วนกลางมาใช้ (ในคำสั่ง *i = 2*w;* ของฟังก์ชัน B) ลักษณะการเชื่อมโยงไปบริเวณคงตัว (static area) เพื่อนำค่าตัวแปรส่วนกลางมาใช้เช่นนี้เรียกว่า *การระบุขอบเขตแบบคงตัว* (static scoping) ซึ่งเป็นลักษณะที่ใช้กันทั่วไปในภาษาคอมพิวเตอร์ปัจจุบัน เช่นภาษา C, C++, Java แต่ในภาษาคอมพิวเตอร์บางภาษาเช่น APL, LISP ใช้วิธี*การระบุขอบเขตแบบพลวัต* (dynamic scoping) ซึ่งหมายถึง เมื่อมีการเรียกใช้ตัวแปรที่ไม่ใช่ตัวแปรเฉพาะที่ จะใช้การติดตามสายเชื่อมโยงพลวัต (dynamic link) ย้อนกลับไปยังฟังก์ชันที่เป็นผู้เรียกไปตามลำดับจนกว่าจะพบชื่อตัวแปรนั้น ดังนั้นในโปรแกรมตัวอย่างข้างต้นถ้าใช้วิธีระบุขอบเขตแบบพลวัต คำสั่ง i=2*w ; ในฟังก์ชัน B จะหมายถึงตัวแปร i ที่ประกาศไว้ในฟังก์ชัน A ไม่ใช่ตัวแปร i ที่ประกาศไว้เป็นตัวแปรส่วนกลาง

<u>วิธีการส่งผ่านพารามิเตอร์</u> (parameter passing methods)

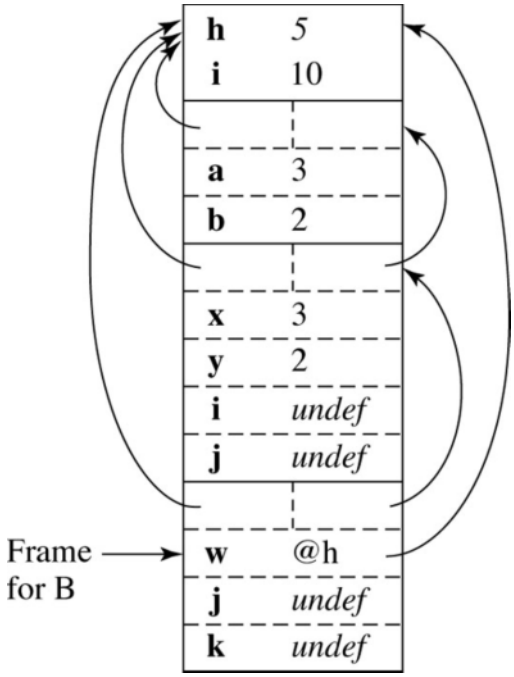
พารามิเตอร์ หรือ อาร์กิวเมนต์ (argument) คือตัวแปรที่ใช้สำหรับการส่งค่าหรือการรับค่า การส่งผ่านพารามิเตอร์จะเกิดขึ้นเมื่อมีการเรียกใช้โปรแกรมย่อย วิธีการส่งผ่านค่าพารามิเตอร์ที่พบบ่อยจะมีสองวิธีคือ *ส่งโดยค่า* (pass by value) และ*ส่งโดยอ้างอิง* (pass by reference) นอกจากนี้ยังมีวิธีการส่งผ่านพารามิเตอร์แบบอื่นๆ อีก คือ *ส่งโดยผลลัพธ์* (pass by result), *ส่งโดยค่าและผลลัพธ์* (pass by value-result), *ส่งโดยชื่อ* (pass by name)

การส่งผ่านพารามิเตอร์ด้วยวิธีส่งโดยค่า (pass by value) เป็นวิธีการสื่อสารระหว่างฟังก์ชันที่ใช้เป็นปกติในภาษา C ดังตัวอย่างในโปรแกรมก่อนหน้านี้ เมื่อฟังก์ชัน main เรียกใช้ ฟังก์ชัน A ด้วยคำสั่ง *A(a,b);* ค่าของ a คือ 3 และค่าของ b คือ 2 จะถูกสำเนา (copy) ส่งไปให้ฟังก์ชัน A ซึ่งตั้งตัวแปรชื่อ x และ y ขึ้นมารับค่า 3 และ 2 ต่อจากนั้นฟังก์ชัน A จะเปลี่ยนแปลงค่า x และ y ไปอย่างไรก็ได้ โดยที่ค่าของ a และ b ในฟังก์ชัน main จะไม่ได้รับผลกระทบใดๆ จากการเปลี่ยนแปลงนั้น

การส่งผ่านพารามิเตอร์ด้วยวิธีส่งโดยอ้างอิง (pass by reference) จะไม่ใช้วิธีการสำเนาค่าตัวแปรส่งไปให้ฟังก์ชันอื่น แต่ใช้วิธีบอกตำแหน่งของตัวแปรเพื่อให้ฟังก์ชันอื่นอ้างอิงตำแหน่งและเข้ามาใช้รวมถึงเปลี่ยนแปลงค่าของตัวแปรนั้นๆได้ จากตัวอย่างโปรแกรมข้างต้นถ้าเปลี่ยนวิธีการส่งพารามิเตอร์ของฟังก์ชัน B เป็นดังต่อไปนี้ (แสดงในรูปแบบของภาษา C++)

```
void  B (int& w)  {
        int j, k ;
        i = 2*w ;
        w = w+1;
        ...
}
```

เมื่อฟังก์ชัน A เรียกใช้ฟังก์ชัน B ด้วยคำสั่ง *B(h);* ผลที่เกิดขึ้นคือตัวแปร w จะกลายเป็นตัวแปรอ้างอิง (reference variable) ที่สามารถอ้างอิงไปยังตำแหน่งของตัวแปร h ดังนั้นการใช้คำสั่ง *w=w+1;* จึงมีผลให้เกิดการเพิ่มค่าของตัวแปร h โครงสร้างภายในหน่วยความจำแสดงได้ดังรูปที่ 4.24



รูปที่ 4.24 โครงสร้างหน่วยความจำเมื่อมีการส่งผ่านพารามิเตอร์ด้วยวิธีส่งโดยอ้างอิง

การส่งผ่านพารามิเตอร์ด้วยวิธีส่งโดยผลลัพธ์ (pass by result) ฟังก์ชันที่เป็นผู้เรียกใช้โปรแกรมย่อยไม่ต้องส่งค่าใดไปให้โปรแกรมย่อย แต่เมื่อโปรแกรมย่อยทำงานเสร็จจะสำเนาค่าส่งกลับมาให้ผู้เรียก

การส่งผ่านพารามิเตอร์ด้วยวิธีส่งโดยค่าและผลลัพธ์ (pass by value-result) ฟังก์ชันที่เป็นผู้เรียกใช้โปรแกรมย่อยจะสำเนาค่าเริ่มต้นส่งไปให้โปรแกรมย่อย และเมื่อโปรแกรมย่อยทำงานเสร็จจะสำเนาค่ากลับมาให้ผู้เรียก

การส่งผ่านพารามิเตอร์ด้วยวิธีส่งโดยชื่อ (pass by name) จะใช้วิธีสำเนาชื่อพารามิเตอร์ส่งไปให้โปรแกรมย่อย และโปรแกรมย่อยจะเปลี่ยนชื่อในคำสั่งให้สอดคล้องกับชื่อพารามิเตอร์ที่รับมา ตัวอย่างเช่น ถ้าโปรแกรมภาษา C++ ต่อไปนี้มีการเรียกใช้โปรแกรมย่อยชื่อ foo และสมมุติให้การส่งผ่านพารามิเตอร์ใช้วิธีส่งโดยชื่อ

```
int MyArray [10];
int foo (NamedVar)  {
        int x=3;
        NamedVar=7;
        return (NamedVar*17);
}
void main ( )  {
        int x=0;
        cout << foo (MyArray [x]);
        cout << foo (x);
}
```

การเรียกใช้โปรแกรมย่อย foo ครั้งแรกด้วยคำสั่ง

cout << foo (MyArray [x]);

จะมีผลให้โปรแกรมย่อย foo รับชื่อ MyArray [x] เป็นพารามิเตอร์ และเปลี่ยนคำว่า NamedVar ในคำสั่งทุกคำสั่งให้เป็นชื่อตามชื่อพารามิเตอร์ ซึ่งจะมีผลให้โปรแกรมย่อย foo เปลี่ยนไปดังนี้

```
int foo (MyArray [x]) {
        int x=3;
        MyArray [x]=7;
        return (MyArray [x]*17);
}
```

การทำงานที่เกิดขึ้นต่อจากนั้นคือ x มีค่าเป็น 3 และ MyArray [3] มีค่า 7 จากนั้นคำนวณค่า 7*17 และมีการส่งค่ากลับเป็น 119 เพื่อแสดงผลออกทางจอภาพด้วยคำสั่ง cout

เมื่อเรียกใช้โปรแกรมย่อย foo ครั้งที่สองด้วยคำสั่ง

cout << foo (x);

จะมีผลให้โปรแกรมย่อย foo รับพารามิเตอร์ x แล้วเปลี่ยน NamedVar ทุกตัวให้เป็นชื่อ x ซึ่งจะทำให้ได้โปรแกรมดังนี้

```
int foo (x) {
        int x=3;
        x =7;
        return (x *17);
}
```

การทำงานที่เกิดขึ้นคือตัวแปร x ที่เป็นตัวแปรเฉพาะที่ภายในโปรแกรมย่อย foo ถูกกำหนดค่าเริ่มต้นเป็น 3 แล้วถูกเปลี่ยนค่าเป็น 7 จากนั้นโปรแกรมย่อยส่งค่ากลับเป็น 119 ซึ่งจะถูกแสดงผลออกทางจอภาพด้วยคำสั่ง cout ในฟังก์ชัน main

การส่งผ่านพารามิเตอร์นอกจากส่งด้วยตัวแปรแล้วภาษาคอมพิวเตอร์หลายภาษายังอนุญาตให้มีการใช้ชื่อฟังก์ชันเป็นพารามิเตอร์ เพื่อส่งผ่านให้โปรแกรมย่อยได้อีกด้วย ดังตัวอย่างภาษา C++ ต่อไปนี้ ที่มีการส่งฟังก์ชัน foo เป็นพารามิเตอร์ไปให้ฟังก์ชัน callfoo (ผลการทำงานที่เกิดขึ้นให้นักศึกษาทดสอบเป็นการบ้าน)

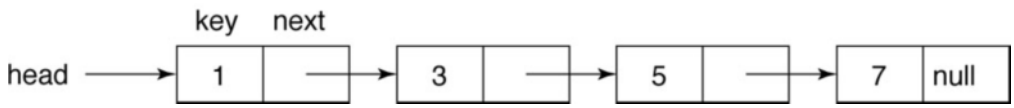
```
#include <iostream.h>
char foo (int x, double y)  {
        cout << x << "   " << y << endl;
        return ('!');
}
void callfoo (char(*fptr) (int, double))   {
      // takes as a parameter a pointer to a function,
      // which in turn takes an int and a double
      // as parameters
         cout << (*fptr) (3,4.5) << endl;
                 // call the passed function
                 // with value 3 and 4.5
}
void  main ( )  {
         callfoo (foo);
                // pass, as a parameter,
                // a pointer to the desired function foo
}
```

<u>โครงสร้างหน่วยความจำกับข้อมูลพ็อยเตอร์</u>

พ็อยเตอร์ (pointer) หรือตัวชี้ เป็นโครงสร้างที่นิยมใช้ในภาษา C, C++, Pascal เพื่อทำหน้าที่เชื่อมโยงไปยังตำแหน่งต่างๆ ในหน่วยความจำ ตัวอย่างต่อไปนี้แสดงการใช้พ็อยเตอร์ในภาษา C

```
Struct Node {
        int key;
        Node * next;
}
Node * head;
```

Node จะหมายถึงโครงสร้างที่ประกอบด้วยข้อมูลสองส่วน ส่วนแรกชื่อ key ทำหน้าที่เก็บค่าเลขจำนวนเต็ม ส่วนที่สองชื่อ next ทำหน้าที่เก็บตัวชี้ หรือพ็อยเตอร์ (แทนด้วยเครื่องหมาย *) ที่ชี้ไปยัง Node ต่อไป ชื่อตัวแปร head จะเป็นชื่อพ็อยเตอร์ที่ชี้ไปยัง Node เริ่มต้น โครงสร้างนี้แสดงเป็นภาพได้ดังรูปที่ 4.25



รูปที่ 4.25 โครงสร้างของลิงค์ลิสต์ที่ประกอบขึ้นจาก Node

ตัวอย่างข้างต้นเป็นพ็อยเตอร์ที่ชี้ไปยังโครงสร้าง Node ที่ถูกจัดสรรเนื้อที่ในหน่วยความจำส่วนที่เรียกว่าฮีพ (heap) ในภาษา C พ็อยเตอร์สามารถชี้ไปยังเนื้อที่เก็บข้อมูลทั้งในส่วนฮีพและส่วนสแตก ดังตัวอย่างต่อไปนี้

```
int *q = malloc(4);          /*q points into heap*/
...
foo ( ) {
     int x;
     int *p = &x;            /* p points into stack*/
  }
```

การใช้พ็อยเตอร์ช่วยให้การเขียนโปรแกรมทำได้สะดวก (writability) แต่ปัญหาที่อาจเกิดขึ้นจากการใช้พ็อยเตอร์ประกอบด้วย

- การใช้พ็อยเตอร์ก่อให้เกิดกรณีการใช้นามแฝง (aliasing) ซึ่งจะทำให้โปรแกรมอ่านยากขึ้น (reduce readability)

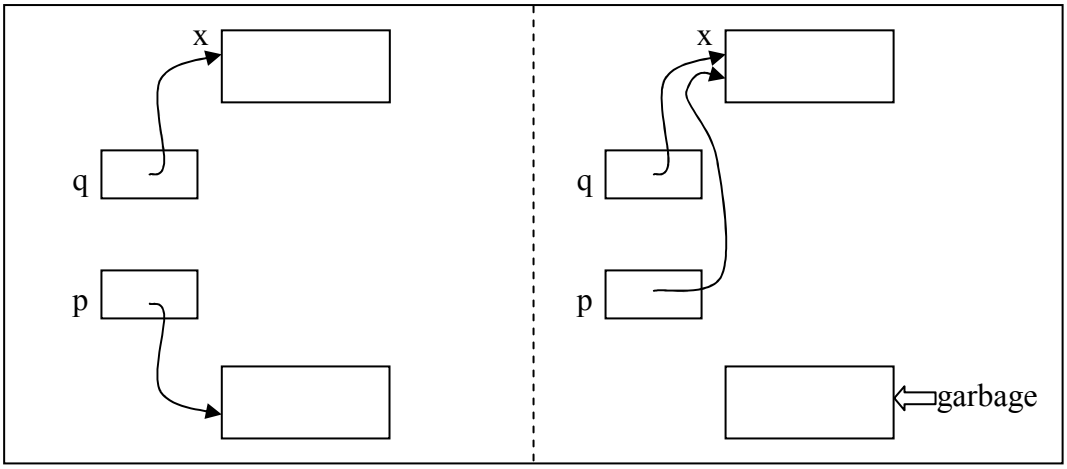
```
int  x,y;
int *p = &x ;          /*    *p is now an alias for x      */
x = 3;
y = *p;
```

- การเพิ่มหรือลดค่าพ็อยเตอร์ อาจทำให้เกิดข้อผิดพลาดซึ่งจะทำให้โปรแกรมมีความน่าเชื่อถือต่ำลง (reduce reliability)

```
int  x;
int *p = &x ;                /*   *p is now an alias for x */
p+= 400;                     /*  Where does p point?  */
```

- การใช้พ็อยเตอร์อาจทำให้เกิดเนื้อที่ ที่ไม่สามารถใช้งาน (garbage)

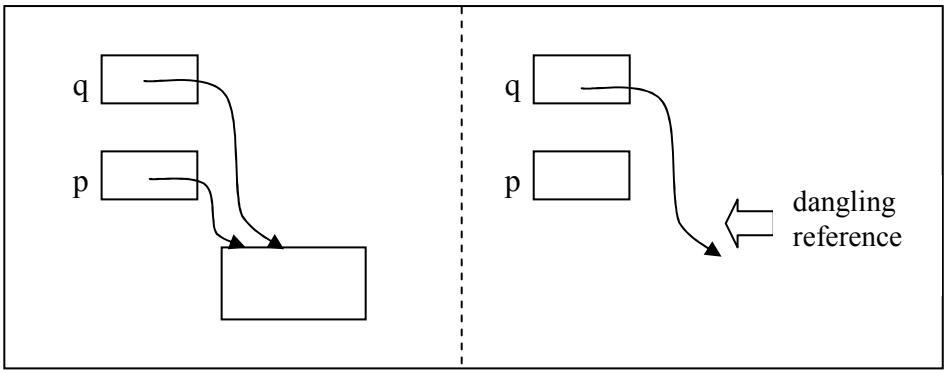
```
int  x;
int *p, *q;                  /*  p and q are pointers to integer*/
q = &x;
p = malloc (sizeof(int));    /*  allocate memory for an int  */
p = q;                       /*  p points to other area, and causes
                                 a garbage  */
```



รูปที่ 4.26 การเกิด garbage จากการเปลี่ยนตำแหน่งชี้ของพ็อยเตอร์

- การใช้พ็อยเตอร์อาจทำให้เกิดการชี้ไปยังเนื้อที่ ที่ถูกยกเลิกการใช้งาน (เรียกว่า dangling pointer หรือ dangling reference)

```
int *p, *q;                  /* p and q are pointers to integers */
...
p = malloc (sizeof(int));    /* allocate memory for an int */
q = p;                       /* p and q point to allocated space */
free(p);                     /* dangling reference in q*/
```



รูปที่ 4.27 การเกิด dangling reference จากการสั่งยกเลิกการใช้งานเนื้อที่ในหน่วยความจำ

<u>โครงสร้างหน่วยความจำกับข้อมูลอาร์เรย์</u>

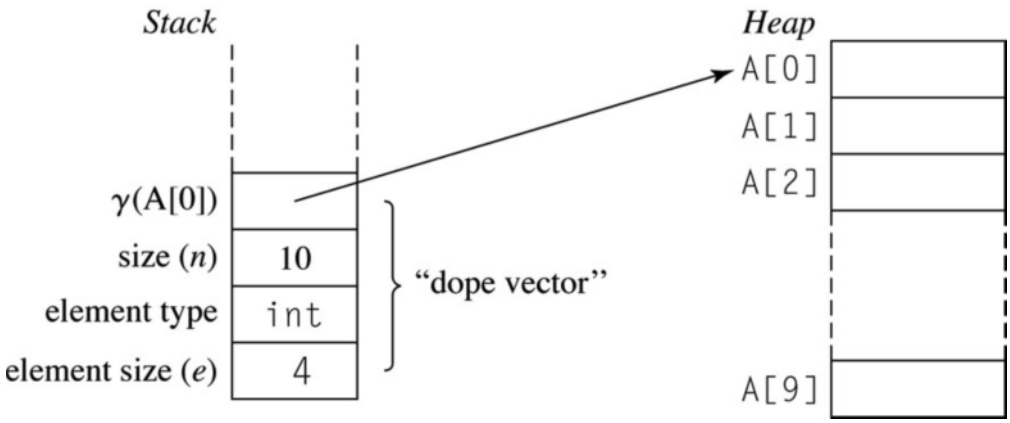
อาร์เรย์ (array) หรือแถวลำดับ เป็นโครงสร้างข้อมูลที่เก็บข้อมูลประเภทเดียวกันไว้เป็นจำนวนมาก ข้อมูลแต่ละตัวจะมีลำดับที่อยู่ที่แน่นอน ตัวอย่างต่อไปนี้แสดงการประกาศโครงสร้างข้อมูลอาร์เรย์ในรูปแบบภาษา Java

```
int [ ] A = new int [10];                  //  an array of 10 int values
float [ ] [ ] [ ] B = new float [3] [4] [2] ;   // a 3x4x2 array of 24 floats
char [ ] [ ] C = new char [4] [3] ;        //a 4x3 array of 12 char values
Object [ ] D = new Object [8];             // an array of 8 object references
```

จากตัวอย่าง A และ D เป็นอาร์เรย์หนึ่งมิติ C เป็นอาร์เรย์สองมิติ และ B เป็นอาร์เรย์สามมิติ ในภาษา Java อาร์เรย์จะได้รับจัดสรรเนื้อที่จากหน่วยความจำฮีพ และถ้าโปรแกรมเมอร์ไม่ได้กำหนดค่าเริ่มต้นให้กับอาร์เรย์ ระบบจะกำหนดค่าให้โดยอัตโนมัติ ข้อมูลประเภทตัวเลข (int, float, short, long, double) จะกำหนดค่าเริ่มต้นเป็นศูนย์ ข้อมูลประเภท char จะกำหนดค่าเริ่มต้นเป็นอักขระว่างซึ่งมีรหัสยูนิโค้ดเป็น '\u0000' ข้อมูลประเภท boolean กำหนดค่าเริ่มต้นเป็นค่า false และข้อมูลประเภท Object จะกำหนดค่าอ้างอิงเริ่มต้นเป็นค่า null

การจัดสรรเนื้อที่หน่วยความจำให้กับโครงสร้างข้อมูลอาร์เรย์ของภาษา Java จะใช้เนื้อที่จากส่วนฮีพ และมีการบันทึกข้อมูลที่เกี่ยวข้องกับอาร์เรย์ไว้ในสแตก (รูปที่ 4.28) ข้อมูลที่เกี่ยวข้องนี้เรียกว่า โดพเวกเตอร์ (dope vector) ประกอบด้วยตำแหน่งแรกของอาร์เรย์ (แทนด้วยสัญลักษณ์ $\gamma(A[0])$ ตามในรูปที่

4.28), ขนาดของอาร์เรย์, ชนิดของข้อมูลในอาร์เรย์, และขนาดของข้อมูลแต่ละตัว (เช่น int ในภาษา Java มีขนาด 4 ไบต์)



รูปที่ 4.28 โครงสร้างหน่วยความจำที่จัดสรรให้กับข้อมูลอาร์เรย์หนึ่งมิติ

ข้อมูลในโดพเวกเตอร์ใช้ประโยชน์ในการตรวจสอบความถูกต้องของคำสั่งในโปรแกรม เช่น ถ้ามีการใช้คำสั่ง A[11] คอมไพเลอร์สามารถตรวจพบความผิดพลาดได้ทันทีและแจ้งข้อความเตือน "Array index out of bound" หรือถ้ามีการกำหนดค่า A[9] = 'x' จะมีการเตือนข้อผิดพลาดว่า 'Type mismatch" เป็นต้น นอกจากนี้ข้อมูลในโดพเวกเตอร์ยังช่วยให้การเข้าถึงตำแหน่งต่างๆ ในอาร์เรย์ทำได้รวดเร็วด้วยการคำนวณดังนี้

$$\gamma(A[i]) = \gamma(A[0]) + (e*i)$$

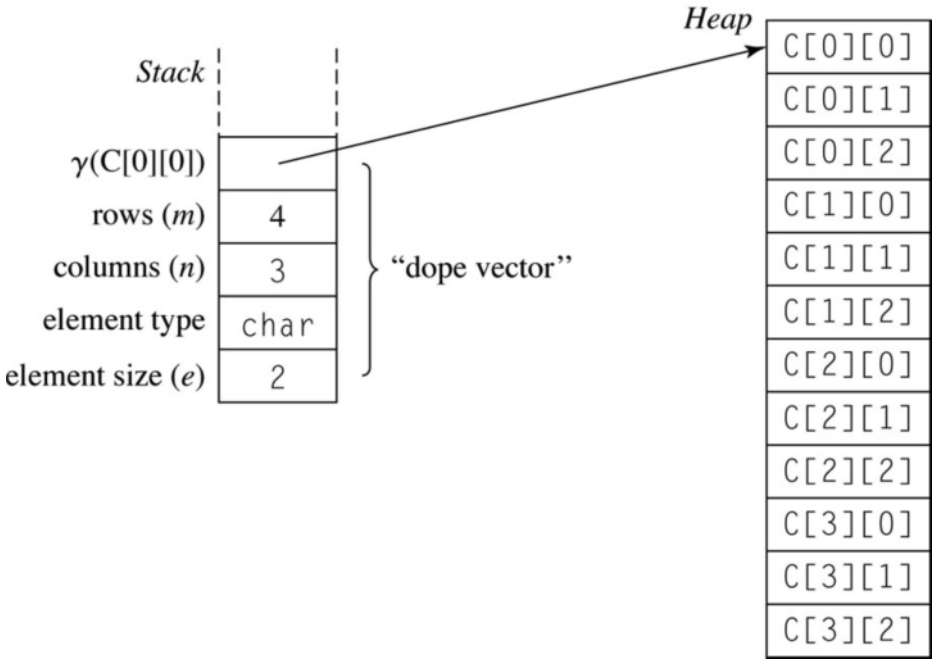
เช่น ตำแหน่งของ A[2] สามารถคำนวณได้จากสูตร

$$\gamma(A[2]) = \gamma(A[0]) + 8$$

ในกรณีของอาร์เรย์หลายมิติ เช่น อาร์เรย์ C ที่เป็นสองมิติ

```
char [ ] [ ] C = new char [4] [3];
```

โครงสร้างหน่วยความจำที่จัดสรรให้กับอาร์เรย์ C แสดงได้ดังรูปที่ 4.29



รูปที่ 4.29 โครงสร้างหน่วยความจำที่จัดสรรให้กับข้อมูลอาร์เรย์สองมิติ

การจัดโครงสร้างหน่วยความจำตามรูปที่ 4.29 เป็นการจัดในแนวแถว (row major order) คือเรียงเนื้อที่หน่วยความจำไปทีละแถว และการคำนวณตำแหน่งของข้อมูลในอาร์เรย์ C สามารถทำได้โดยใช้สูตร

$$\gamma\,(C[i][j]) = \gamma\,(C[0][0]) + e(n*i + j)$$

เช่น ถ้าต้องการคำนวณตำแหน่งของ C[2][1] สามารถทำได้ดังนี้

$$\gamma\,(C[2][1]) = \gamma\,(C[0][0]) + 2(3*2+1)$$

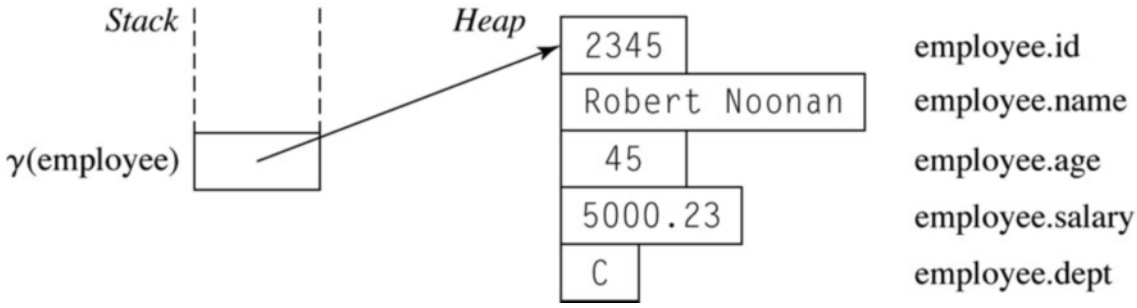
$$= \gamma\,(C[0][0]) + 14$$

<u>โครงสร้างหน่วยความจำกับข้อมูลเรคคอร์ด</u>

ข้อมูลเรคคอร์ดหรือข้อมูล structure เป็นโครงสร้างข้อมูลที่มีข้อมูลหลายประเภทปะปนกัน ดังตัวอย่างต่อไปนี้

รูปแบบภาษา C	**รูปแบบภาษา Java**
typedef struct { *int id;* *char name[25];* *int age;* *float salary;* *char dept;* *} employeeType;* *employeeType employee;*	*class employeeType {* *int id;* *String name =new String [25]* *int age;* *float salary;* *char dept;* *}* *employeeType employee;*

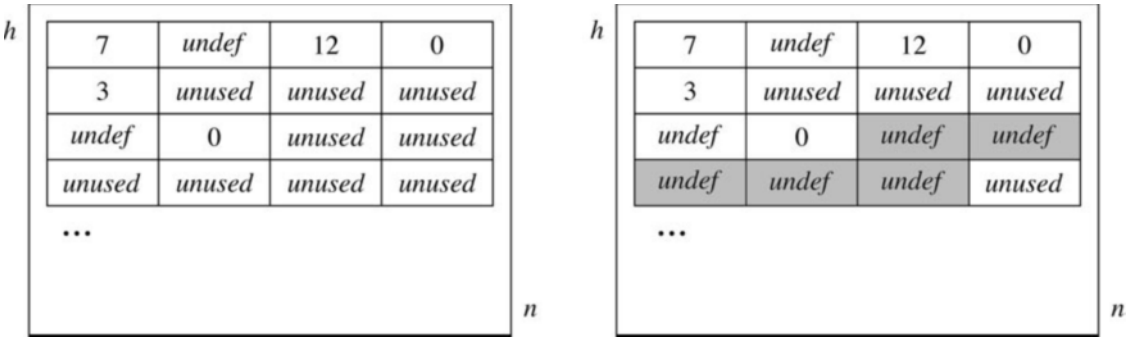
การจัดสรรเนื้อที่หน่วยความจำให้กับข้อมูลเรคคอร์ด จะใช้เนื้อที่ส่วนฮีพ (ดังรูปที่ 4.30) โดย γ(employee) ในสแตกจะทำหน้าที่ชี้ตำแหน่งเริ่มต้นของเรคคอร์ด



รูปที่ 4.30 โครงสร้างหน่วยความจำของข้อมูลเรคคอร์ด

<u>การจัดสรรเนื้อที่หน่วยความจำฮีพ</u>

เมื่อมีการเรียกใช้หน่วยความจำฮีพ (เช่นเมื่อมีการเรียกใช้คำสั่ง new) ระบบจะค้นหาเนื้อที่ว่างจากฮีพที่มีขนาดใกล้เคียงกับเนื้อที่ที่ต้องการใช้งาน เนื้อที่นี้จะอยู่ตรงจุดใดของฮีพก็ได้ ซึ่งจะแตกต่างจากการจัดสรรหน่วยความจำสแตกที่มีการจัดสรรเนื้อที่จากปลายด้านเดียวของสแตกเท่านั้น รูปที่ 4.31 แสดงโครงสร้างของฮีพเมื่อมีการใช้คำสั่ง new(5) เพื่อเรียกใช้เนื้อที่ 5 บล็อกจากฮีพ ระบบจะค้นหาเนื้อที่ที่ประกาศว่ายังไม่มีการใช้งาน (unused) เพื่อเปลี่ยนให้เป็นเนื้อที่ใช้งาน โดยถ้ายังไม่มีการกำหนดค่าเริ่มต้นจะประกาศว่ายังไม่กำหนดค่า (undef) จากนั้นจะส่งตำแหน่งของบล็อกเริ่มต้นไปให้กับคำสั่ง new (ตามรูปตัวอย่างจะส่งตำแหน่ง h+10) แต่ถ้าในขณะนั้นเนื้อที่ว่างในฮีพมีไม่ถึง 5 บล็อก ระบบจะบอกข้อความผิดพลาดว่า ฮีพเต็ม (heap overflow) และคำสั่ง new จะมีสถานะล้มเหลว

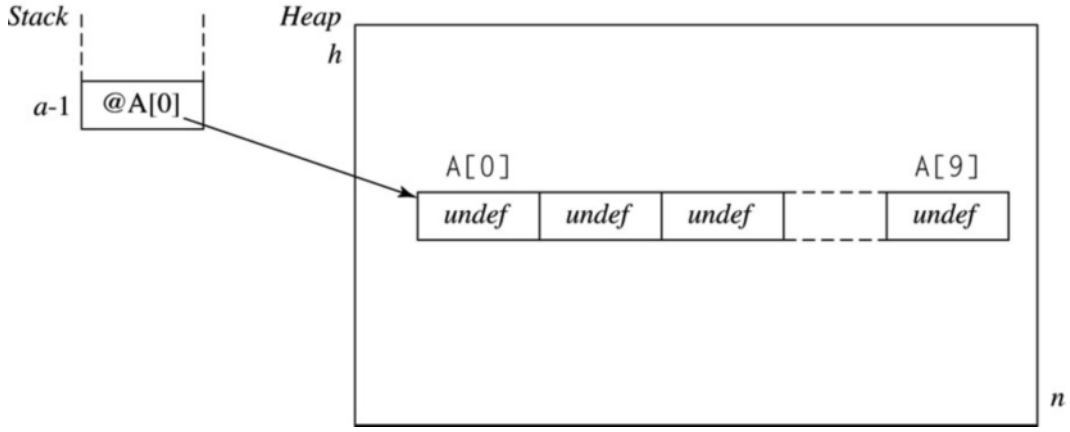


รูปที่ 4.31 โครงสร้างหน่วยความจำฮีพก่อนและหลังการใช้คำสั่ง new(5)

ในกรณีการจัดสรรเนื้อที่ฮีพให้กับข้อมูลอาร์เรย์

int[ ] A = new int[10];

แสดงโครงสร้างฮีพได้ดังรูปที่ 4.32

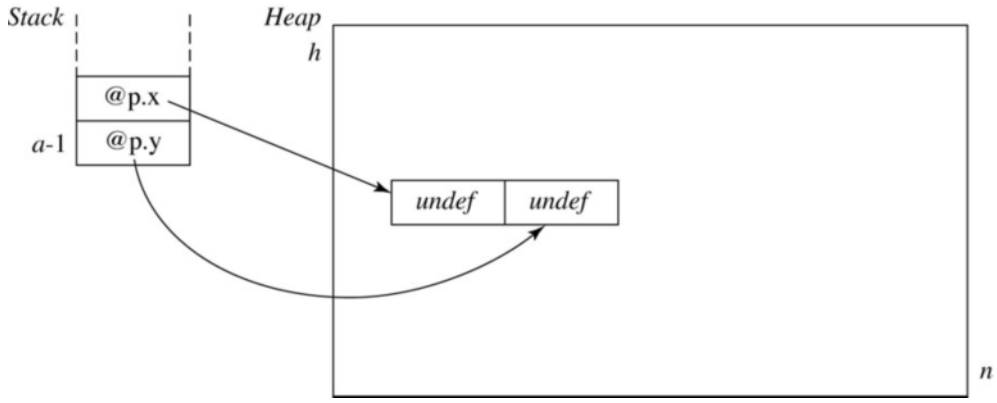


รูปที่ 4.32 การจัดสรรหน่วยความจำฮีพให้กับอาร์เรย์หนึ่งมิติ

การจัดสรรหน่วยความจำในกรณีของการใช้ struct หรือ เรคคอร์ด

```
struct point {
        int x;
        int y;
}
point p = new point( );
```

แสดงโครงสร้างหน่วยความจำฮีพได้ดังรูปที่ 4.33



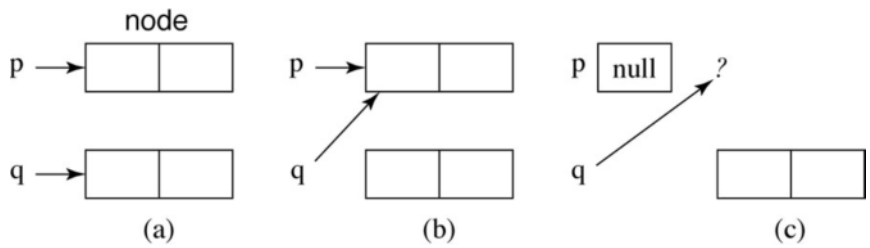
รูปที่ 4.33 การจัดสรรหน่วยความจำฮีพให้กับโครงสร้างข้อมูล struct

<u>การจัดการหน่วยความจำฮีพ</u>

ภาษา Java ที่เป็นภาษาเชิงวัตถุ และภาษาเชิงวัตถุอื่นๆ มีการใช้งานหน่วยความจำฮีพอย่างมากเพื่อเก็บข้อมูลอ็อบเจ็ค และเมื่ออ็อบเจ็คถูกยกเลิกการใช้งานเนื้อที่ส่วนนั้นจะกลายเป็นส่วนที่เรียกว่า ขยะ หรือ garbage ดังตัวอย่างต่อไปนี้

```
class node {
      int value;
      node next;
}
...
node p, q;
p = new node( );
q = new node( );
```

ข้อความสั่งข้างต้นเป็นการสั่งสร้างโหนดเพื่อใช้ในลิงค์ลิสต์ ซึ่งโครงสร้างการจัดสรรเนื้อที่หน่วยความจำฮีพ จะเป็นดังรูปที่ 4.34 (a)



รูปที่ 4.34 ตัวอย่างการเกิด garbage และ dangling reference

ต่อมาถ้ามีการใช้คำสั่ง

q = p;

ซึ่งจะมีผลให้ q เปลี่ยนการชี้ให้ไปชี้ที่เดียวกับ p (รูปที่ 4.34(b)) พื้นที่เดิมที่ q เคยชี้อยู่จะกลายเป็นพื้นที่ที่โปรแกรมไม่สามารถเข้าถึงและใช้งานได้เรียกพื้นที่นี้ว่าขยะ (garbage) และถ้ามีการใช้คำสั่งต่อไปนี้เพิ่มเติม

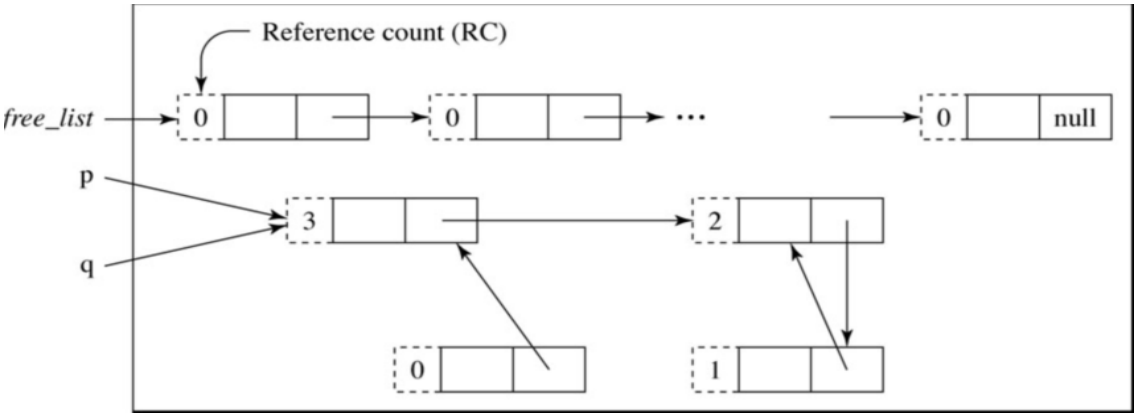
delete(p);

ซึ่งเป็นการสั่งยกเลิกใช้งานพื้นที่ ที่ p เคยชี้อยู่ ผลที่เกิดขึ้นคือ q ชี้ไปยังพื้นที่ที่ถูกยกเลิกการใช้งาน (รูปที่ 4.34(c)) เหตุการณ์เช่นนี้จะเรียกว่า q เป็นตัวชี้ลอย (dangling reference)

เพื่อป้องกันการเกิดเหตุการณ์ตัวชี้ลอยและเพื่อการใช้งานฮีพอย่างมีประสิทธิภาพ (ไม่ให้มีพื้นที่ขยะค้างอยู่ในฮีพ) ภาษาคอมพิวเตอร์ที่มีการใช้เนื้อที่ฮีพจำนวนมากอย่างเช่น ภาษา Java และภาษา LISP จึงมีโปรแกรมพิเศษที่ทำหน้าที่จัดเก็บขยะ (garbage collection) ให้โดยอัตโนมัติ วิธีจัดเก็บขยะที่นิยมใช้จะมีอยู่ 3 แนวทางหลัก คือ การนับตัวชี้ (reference counting), การทำเครื่องหมาย (mark-sweep), และการรวบรวมสำเนา (copy collection)

<u>การจัดเก็บขยะด้วยวิธีการนับตัวชี้</u>

วิธีการจัดการฮีพแบบนี้ จะเริ่มต้นด้วยการเชื่อมโยงเนื้อที่ว่างทั้งหมดในฮีพไว้ในลิงค์ลิสต์ชื่อ free_list (รูปที่ 4.35) ทุกโหนดหรือทุกบล็อกของฮีพจะมีส่วนที่เก็บค่าตัวนับเรียกว่า reference count(RC) บล็อกที่ยังไม่ถูกใช้งานค่า RC จะเป็นศูนย์ เมื่อมีการเรียกใช้งานค่า RC จะถูกบวกเพิ่มขึ้นเท่ากับจำนวนตัวชี้ที่ชี้อ้างอิงเข้ามาที่พื้นที่บล็อกนั้น และเมื่อมีการลดจำนวนตัวชี้ค่า RCของบล็อกจะถูกลดค่าลงตามไปด้วย บล็อกใดที่มีค่า RC เป็นศูนย์ จะถูกดึงกลับไปรวมอยู่ในสายของ free_list

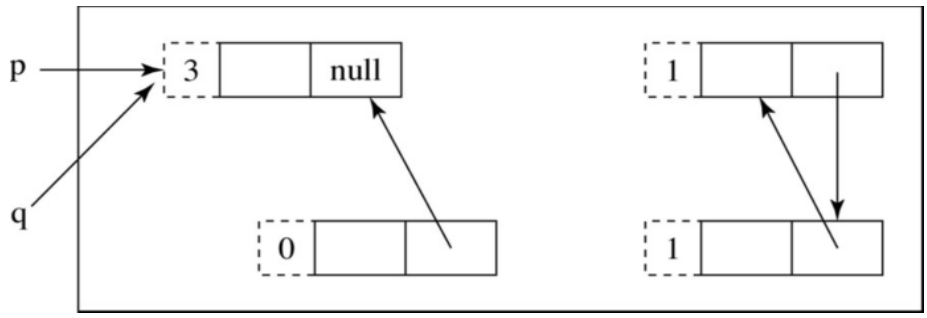


รูปที่ 4.35 ตัวอย่างโครงสร้างฮีพเมื่อใช้การจัดเก็บขยะด้วยวิธีการนับตัวชี้

ข้อดีของวิธีการนี้คือจัดการง่าย แต่ข้อเสียคือต้องเสียพื้นที่ต้นบล็อกเพื่อเก็บค่า RC และในบางครั้งบล็อกที่เก็บขยะ(ไม่ได้ถูกใช้งานแล้ว) แต่ค่า RC ไม่เป็นศูนย์ จะไม่ถูกจัดเก็บไปไว้ใน free_list เช่นตัวอย่างจากรูปที่ 4.35 เมื่อมีการใช้คำสั่ง

p -> next = null;

จะมีผลให้บล็อกข้อมูล 2 บล็อก ทางขวามือของรูปที่ 4.36 กลายเป็นขยะ แต่ค่า RC ไม่เป็นศูนย์



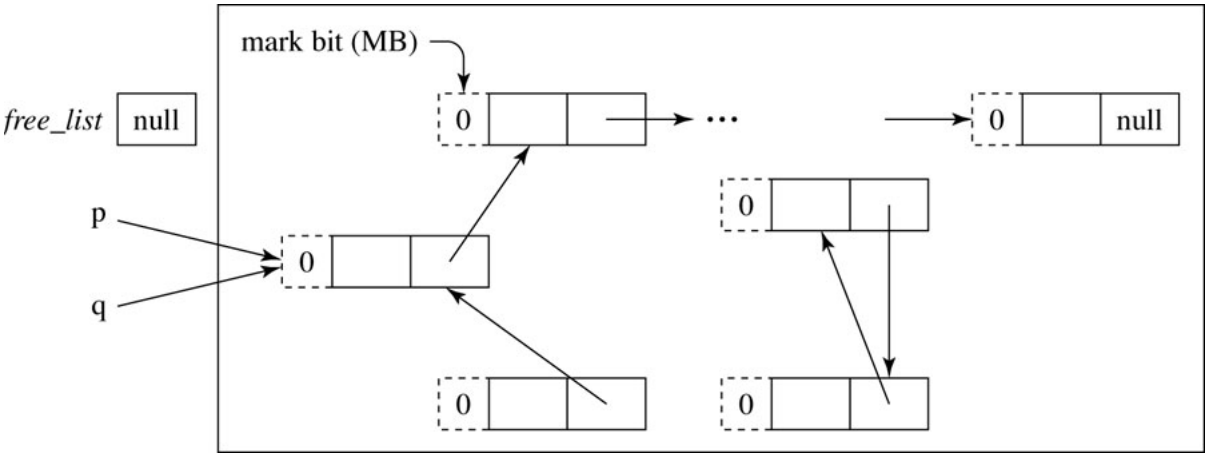
รูปที่ 4.36 การเกิดขยะโดยจำนวนตัวชี้มีค่ามากกว่าศูนย์

<u>การจัดเก็บขยะด้วยวิธีทำเครื่องหมาย</u>

โปรแกรมจัดเก็บขยะด้วยวิธีทำเครื่องหมายนี้ จะทำงานเมื่อเกิดเหตุการณ์ฮีพเต็ม นั่นคือเมื่อ free_list มีค่าเป็น null ซึ่งจะแตกต่างจากโปรแกรมจัดเก็บขยะด้วยวิธีนับตัวชี้ ที่จะทำงานทุกครั้งเมื่อมีการกำหนดพ็อยเตอร์ หรือเมื่อมีการเปลี่ยนแปลงการชี้ของพ็อยเตอร์

การจัดเก็บขยะด้วยวิธีทำเครื่องหมายจะแบ่งการทำงานเป็นสองช่วง ช่วงแรกเป็นช่วงทำเครื่องหมาย (mark phase) ช่วงที่สองเป็นช่วงกวาดขยะ (sweep phase)

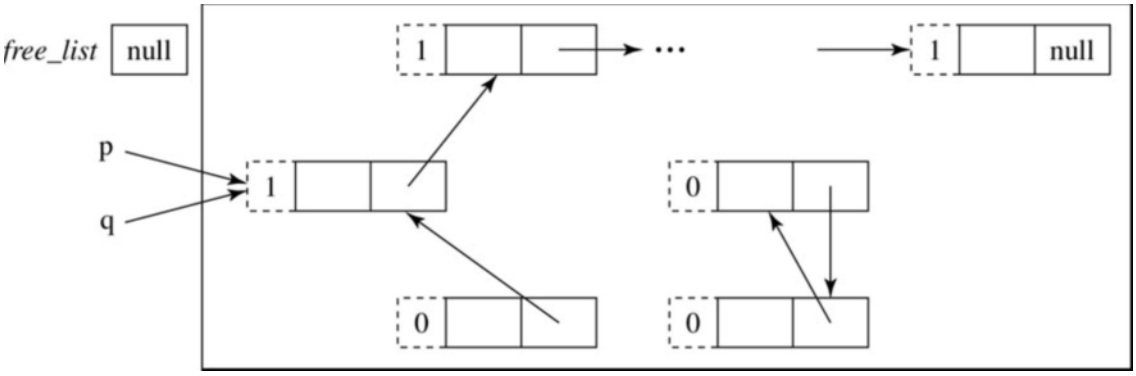
โครงสร้างของบล็อกทุกบล็อกในฮีพ จะมีบิตพิเศษที่ต้นบล็อกเรียกว่า บิตเครื่องหมาย (mark bit) ซึ่งปกติจะมีค่าเป็นศูนย์ เมื่อ free_list มีค่าเป็น null จะเป็นกรณีที่ฮีพเต็ม (แสดงภาพตัวอย่างได้ดังรูปที่ 4.37)



รูปที่ 4.37 ตัวอย่างโครงสร้างบล็อกในฮีพเมื่อใช้การจัดเก็บขยะด้วยวิธีทำเครื่องหมาย

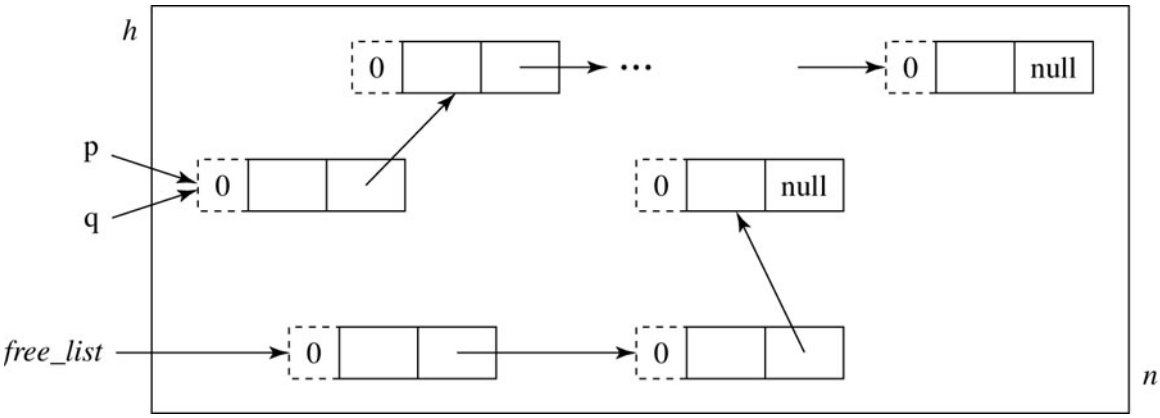
กระบวนการจัดเก็บขยะจะเริ่มด้วยช่วงทำเครื่องหมาย โดยโปรแกรมจะตรวจสอบตัวแปรทั้งหมดในสแตกว่ามีตัวแปรใดบ้างที่อ้างอิงมายังพื้นที่ในฮีพ ในภาพตัวอย่างสมมุติให้มีเพียงตัวแปร p และ q ที่ชี้มายังฮีพ โปรแกรมจะตามตัวชี้ p และ q มายังพื้นที่ในฮีพ แล้วทำเครื่องหมายในบล็อกแรกโดยเปลี่ยนบิตเครื่องหมายจากศูนย์เป็นหนึ่ง จากนั้นตามตัวชี้จากบล็อกแรกไปยังบล็อกต่อไปเพื่อทำบิตเครื่องหมายหนึ่งไล่ไปตามลำดับจนครบทุกตัวแปรและทุกบล็อกที่สามารถเชื่อมโยงไปได้ รูปที่ 4.38 แสดงเหตุการณ์หลังจาก

เสร็จสิ้นช่วงทำเครื่องหมาย พื้นที่ในฮีพทุกบล็อกที่ยังมีการใช้งานอยู่บิตเครื่องหมายจะเป็นหนึ่ง ในขณะที่บล็อกที่ไม่มีการใช้งานแล้วบิตเครื่องหมายจะเป็นศูนย์



รูปที่ 4.38 โครงสร้างบล็อกในฮีพ หลังเสร็จสิ้นช่วงทำเครื่องหมาย

ในช่วงที่สองที่เป็นช่วงกวาดขยะโปรแกรมจะตรวจสอบทุกบล็อกในฮีพ บล็อกใดที่บิตเครื่องหมายเป็นศูนย์ จะถูกเชื่อมโยงเข้ากับ free_list เพื่อให้สามารถนำพื้นที่กลับไปใช้งานใหม่ได้ เมื่อเสร็จการเชื่อมโยง free_list แล้วบิตเครื่องหมายของทุกบล็อกจะถูกเปลี่ยนกลับเป็นศูนย์ (รูปที่ 4.39)

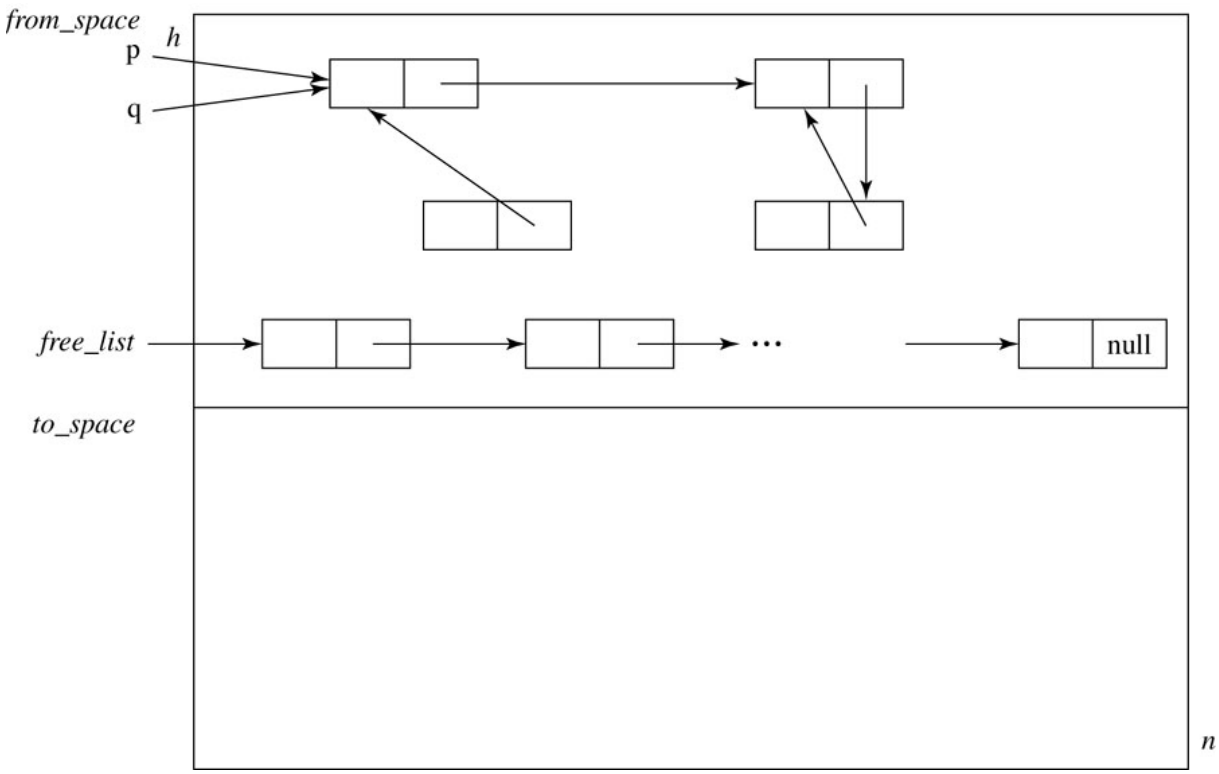


รูปที่ 4.39 โครงสร้างในฮีพหลังเสร็จสิ้นช่วงกวาดขยะ

วิธีการจัดเก็บขยะแบบนี้มีข้อดีที่ใช้พื้นที่เก็บเครื่องหมายพิเศษเพียงหนึ่งบิตต่อบล็อก และทุกบล็อกที่ไม่มีการใช้งานจะถูกเรียกคืนมารวมไว้ใน free_list แต่ข้อเสียคือ ขณะที่โปรแกรมจัดเก็บขยะทำงานโปรแกรมอื่นๆ จะต้องหยุดรอจนกว่ากระบวนการจัดเก็บขยะจะเสร็จสิ้น

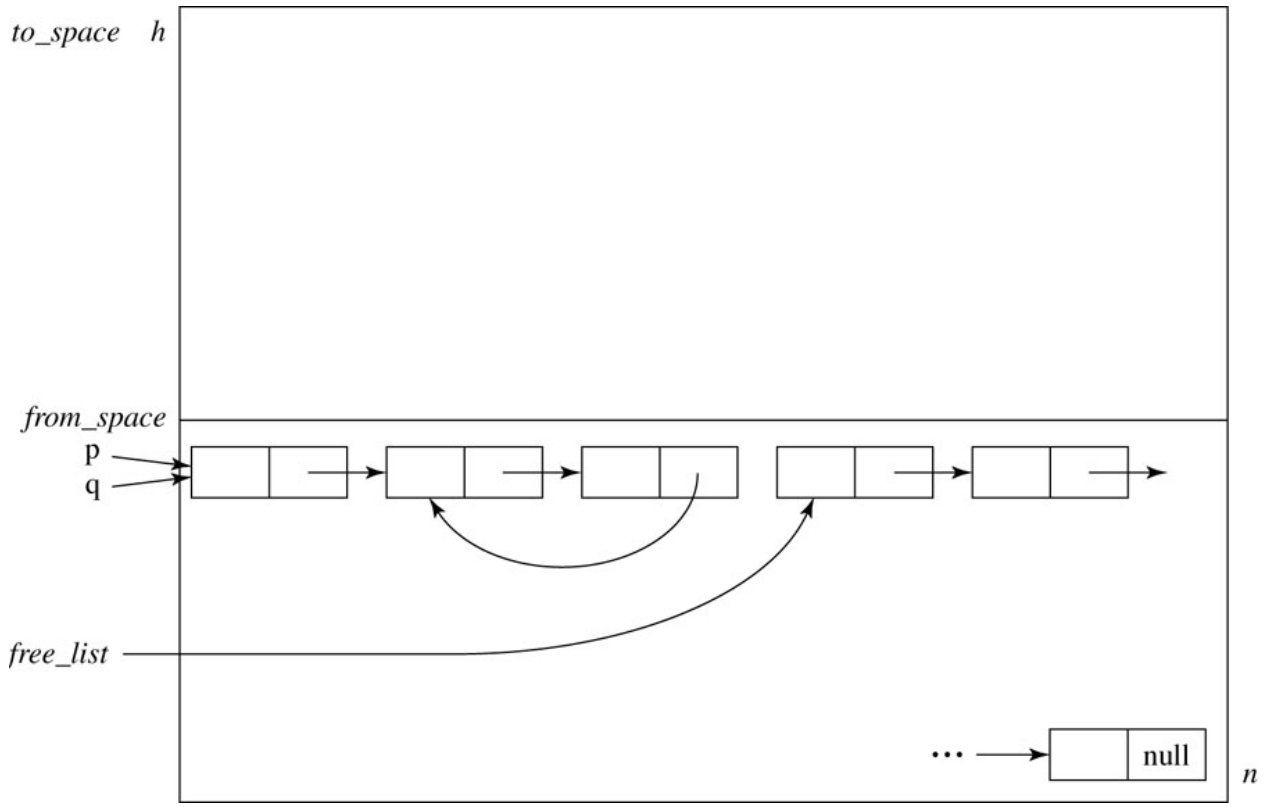
<u>การจัดเก็บขยะด้วยวิธีการรวบรวมสำเนา</u>

วิธีการนี้ไม่จำเป็นต้องมีเนื้อที่พิเศษที่ต้นบล็อก แต่จะใช้วิธีแบ่งเนื้อที่ฮีพออกเป็นสองส่วน เรียกว่า ส่วน from_space และส่วน to_space (รูปที่ 4.40)



รูปที่ 4.40 โครงสร้างเริ่มต้นของฮีพก่อนการจัดเก็บขยะด้วยวิธีการรวบรวมสำเนา

เมื่อฮีพเต็มหรือเมื่อมีการเรียกใช้โปรแกรมจัดเก็บขยะ โปรแกรมจะตรวจสอบบล็อกในส่วน from_space บล็อกใดที่ยังมีการใช้งานอยู่จะถูกย้ายไปอยู่ในส่วน to_space บล็อกใดที่ไม่มีการใช้งานแล้ว จะถูกย้ายไปอยู่ในส่วน to_space เช่นเดียวกันแต่อยู่ในกลุ่มเชื่อมโยงของ free_list เมื่อย้ายบล็อกทั้งหมดแล้ว จะสลับชื่อจาก to_space เป็น from_space และ from_space เปลี่ยนชื่อเป็น to_space (ดังรูปที่ 4.41)



รูปที่ 4.41 โครงสร้างฮีพหลังการจัดเก็บขยะด้วยวิธีการรวบรวมสำเนา

4.5 สรุป

การแปลภาษาเป็นขั้นตอนที่สำคัญ ที่จะช่วยให้โปรแกรมที่เขียนอยู่ในรูปแบบของภาษาระดับสูงสามารถประมวลผลได้บนเครื่องคอมพิวเตอร์ที่รู้จักเฉพาะภาษาเครื่อง โปรแกรมที่ใช้ในการแปลภาษาแบ่งออกได้เป็นสองประเภท คือ อินเตอร์พรีตเตอร์ และคอมไพเลอร์ อินเตอร์พรีตเตอร์จะใช้วิธีแปลคำสั่งในภาษาระดับสูงทีละคำสั่งพร้อมทั้งประมวลผลตามคำสั่งนั้นทันที ขั้นตอนการทำงานของอินเตอร์พรีตเตอร์จึงไม่ซับซ้อน และสามารถรายงานข้อผิดพลาดที่พบได้ตรงจุด

คอมไพเลอร์มีขั้นตอนการทำงานที่ซับซ้อนมากกว่าอินเตอร์พรีตเตอร์ ทั้งนี้เพื่อความสะดวกในการแปลแยกส่วนและเพื่อช่วยให้สามารถปรับปรุงคำสั่งให้สามารถประมวลผลได้อย่างมีประสิทธิภาพมากขึ้น นอกจากนี้การแปลภาษาของคอมไพเลอร์จะมีการสร้างรหัสจุดหมาย (object code) ที่สามารถนำไปประมวลผลซ้ำได้หลายครั้งโดยไม่ต้องผ่านขั้นตอนการแปลอีก

พัฒนาการของการแปลภาษาในระยะหลังที่มีการสร้างเครื่องจักรเสมือนซึ่งคือ การเขียนซอฟต์แวร์ขึ้นเพื่อจำลองให้ทำหน้าที่เสมือนฮาร์ดแวร์ ทำให้สามารถสร้างรหัสจุดหมายที่ประมวลผลบนเครื่องระบบใดก็ได้ ภาษาที่ประสบความสำเร็จในหลักการนี้มากที่สุดคือ ภาษา Java ที่อาศัย Java Virtual Machine (JVM) เป็นเครื่องจักรเสมือนประมวลผลรหัสจุดหมายที่อยู่ในรูปแบบไบต์โค้ด

การทำให้โปรแกรมในภาษาระดับสูงประมวลผลได้บนเครื่องคอมพิวเตอร์ นอกจากโปรแกรมแปลภาษาแล้ว ยังต้องมีโปรแกรมระบบที่เรียกว่า run-time system ทำหน้าที่จัดการกับข้อมูลในหน่วยความจำ ซึ่งเป็นขั้นตอนการประมวลผลโปรแกรมที่ต่อเนื่องจากการแปลโปรแกรม

การทำงานของโปรแกรมส่วนใหญ่จะใช้วิธีการกำหนดค่าให้กับตัวแปรแล้วเปลี่ยนแปลงค่าของตัวแปรไปตามลำดับ และเมื่อสิ้นสุดการทำงานของโปรแกรม ค่าสุดท้ายของตัวแปรจะเป็นตัวบอกผลลัพธ์การทำงานของโปรแกรม

ขณะที่โปรแกรมทำงาน ตัวแปรต่างๆ ของโปรแกรมจะใช้เนื้อที่ในหน่วยความจำหลักส่วนที่ใช้เก็บข้อมูล ซึ่งแยกย่อยออกเป็น 3 บริเวณ คือ บริเวณคงตัว (static area) ใช้เก็บค่าตัวแปรส่วนกลาง, บริเวณสแตก (rut-time stack) ใช้เก็บค่าตัวแปรเฉพาะที่ของแต่ละโปรแกรมย่อย และ บริเวณฮีพ (heap) ใช้เก็บค่าตัวแปรที่มีการจัดสรรเนื้อที่ระหว่างที่โปรแกรมทำงาน เช่นเมื่อมีการใช้คำสั่ง new หรือใช้คำสั่ง malloc

เนื้อที่ส่วนสแตกจะได้รับการจัดสรรเมื่อโปรแกรมย่อยถูกเรียกใช้งานและคืนเนื้อที่เมื่อโปรแกรมย่อยทำงานเสร็จ เนื้อที่ส่วนของฮีพมีการจัดการต่างจากสแตกการเรียกคืนเนื้อที่ในฮีพต้องมีโปรแกรมจัดการโดยเฉพาะ เรียกว่าโปรแกรมทำ garbage collection ทำหน้าที่รวบรวมพื้นที่ที่ไม่ได้ใช้งานแล้วกลับคืนมาใว้ใน free_list

แบบฝึกหัดท้ายบทที่ 4

คำถามปรนัย: ให้เลือกคำตอบที่ถูกต้องที่สุด

1. ข้อใดคือข้อแตกต่างระหว่าง interpreter และ compiler ?
   - ก. interpreter มีขั้นตอนน้อยกว่า
   - ข. compiler เมื่อรับ source program แล้วจะแปลงเป็นภาษาอื่น
   - ค. interpreter ต้องใช้ source program ทุกครั้งในการ run
   - ง. ถูกทุกข้อ

2. ข้อใดเป็นข้อดีของ interpreter ?
   - ก. มีประสิทธิภาพสูงกว่า compiler
   - ข. การ run โปรแกรมเร็วกว่า compiler
   - ค. สามารถแปลแบบแยกส่วนได้
   - ง. ใช้เวลาสร้างได้เร็วกว่า compiler

3. ในการแปลโปรแกรม ขั้นตอน semantic analysis มีหน้าที่ทำอะไร ?
   - ก. วิเคราะห์ความถูกต้องของ token
   - ข. วิเคราะห์ความถูกต้องของไวยากรณ์
   - ค. วิเคราะห์ความถูกต้องของคำสั่ง
   - ง. วิเคราะห์เพื่อปรับปรุงคำสั่ง

4. เครื่องคอมพิวเตอร์เสมือน (virtual machine) คืออะไร ?
   - ก. การใช้ซอฟต์แวร์จำลองการทำงานของฮาร์ดแวร์
   - ข. ส่วนประกอบหนึ่งของการคอมไพล์โปรแกรม Java
   - ค. อุปกรณ์ที่เป็นตัวกลางระหว่างฮาร์ดแวร์และซอฟต์แวร์
   - ง. การใช้เครื่องคอมพิวเตอร์จำลองคอมพิวเตอร์เครื่องอื่น

5. ภาษาคอมพิวเตอร์ต่อไปนี้ภาษาใดเกิดขึ้นก่อน ?
   - ก. FORTRAN I
   - ข. ASSEMBLY
   - ค. COBOL
   - ง. ALGOL

6. ภาษาในระดับต่ำ (low level) มีลักษณะอย่างไร ?
   - ก. การใช้งานมีความซับซ้อนต่ำ เหมาะสำหรับผู้เริ่มต้นเขียนโปรแกรม
   - ข. เป็นภาษาที่มีระดับความซับซ้อนต่ำ ใกล้เคียงกับภาษาเครื่อง
   - ค. จำนวนผู้ใช้งานมีต่ำ ไม่นิยมนำมาเขียนโปรแกรม
   - ง. การลงทุนเพื่อนำภาษามาใช้งานต่ำ ราคาถูก

7. ข้อใดคือต้นแบบของภาษาในการเขียน Pseudo-Code ?
   - ก. BASIC
   - ข. LISP
   - ค. ALGOL
   - ง. PL/I

8. พิจารณาส่วนของโปรแกรมภาษา C ต่อไปนี้

```
int * ptr;
int count, init = 1;
ptr = &init;
count = *ptr;
```

ส่วนของโปรแกรมนี้มีผลการทำงานเทียบเท่ากับชุดคำสั่งในข้อใด ?

ก.
```
int count;
int init = 1;
count = init;
```

ข.
```
int count;
int init = 1;
int * ptr = &init;
count = *ptr;
```

ค.
```
int count,init;
count = init = 1;
```

ง. ถูกทุกข้อ

9. การประกาศใช้ตัวแปร ptr ในข้อ 8 ทำให้เกิดเหตุการณ์ใด ?

ก. alias
ข. dangling pointer
ค. garbage
ง. semantic error

10. pointer ที่ชี้ตำแหน่งไปที่เนื้อที่หน่วยความจำ heap ซึ่งเคยใช้เก็บค่าตัวแปร แต่ปัจจุบันตัวแปรนั้นถูกยกเลิกการใช้งานไปแล้ว pointer แบบนี้เรียกว่าอะไร ?

ก. free pointer
ข. lost pointer
ค. leakage pointer
ง. dangling pointer

11. ภาษา Java หลีกเลี่ยงปัญหาที่เกิดจากการใช้ pointer ด้วยวิธี ?

ก. เปลี่ยนไปใช้ reference type
ข. ให้ใช้เฉพาะ class ที่ผู้ใช้สร้างขึ้นเอง
ค. ยกเลิกการใช้ค่า nil และ null
ง. ให้ใช้คำสั่ง new

12. พารามิเตอร์ที่กำหนดไว้ที่ส่วนหัวของฟังก์ชัน เรียกว่าอะไร ?

ก. prototype
ข. header parameter
ค. formal parameter
ง. actual parameter

13. พารามิเตอร์ที่ปรากฏในคำสั่งเรียกใช้ฟังก์ชัน เรียกว่าอะไร ?

ก. prototype parameter
ข. header parameter
ค. formal parameter
ง. actual parameter

14. ถ้าพารามิเตอร์ถูกส่งด้วยวิธี pass-by-value จะมีการทำงานใดเกิดขึ้น ?

ก. actual parameter จะถูกเปลี่ยนชื่อให้ตรงกับตัวแปรในฟังก์ชัน
ข. actual parameter จะถูกเปลี่ยนเป็นตัวชี้ตำแหน่งในหน่วยความจำ
ค. formal parameter จะทำหน้าที่เป็นตัวแปร local ในฟังก์ชัน
ง. formal parameter จะถูกเปลี่ยนชื่อให้ตรงกับ actual parameter

15. ข้อเสียของวิธีการส่งผ่านค่าพารามิเตอร์แบบ pass-by-value คืออะไร ?

ก. ต้องเสียเนื้อที่หน่วยความจำเพื่อเก็บค่า formal parameter

ข. ต้องเสียเนื้อที่หน่วยความจำเพื่อเก็บค่า actual parameter

ค. ไม่สามารถตรวจสอบชนิดของตัวแปรได้

ง. ไม่สามารถทำ recursive ได้

16. การส่งผ่านพารามิเตอร์แบบใดที่เป็นการส่งค่าตำแหน่งในหน่วยความจำ ?

ก. pass-by-reference ข. pass-by-value

ค. pass-by-value-result ง. pass-by-name

กำหนดซูโดโค้ดที่มีไวยากรณ์คล้ายภาษา C ให้ดังต่อไปนี้ (ใช้ตอบคำถามข้อ 17-19)

```
void  f(int x, int y)  { x = 1; y = 3; }
void  main()  {
      int  i, a[2];
      i = 0;  a[0] = 0;  a[1] = 0;
      f(i, a[i]);
      printf(" %d  %d  %d \n", i , a[0], a[1] );
   }
```

17. ถ้าการส่งผ่านค่าระหว่างฟังก์ชัน main() และฟังก์ชัน f() เป็นแบบ pass-by-reference ผลลัพธ์ที่ได้จากคำสั่ง printf() คือค่าใด ?

ก. 0 0 0 ข. 1 0 3

ค. 1 3 0 ง. 0 1 3

18. ถ้าการส่งผ่านค่าระหว่างฟังก์ชัน main() และฟังก์ชัน f() เป็นแบบ pass-by-name ผลลัพธ์ที่ได้จากคำสั่ง printf() คือค่าใด ?

ก. 0 0 0 ข. 1 0 3

ค. 1 3 0 ง. 0 1 3

19. ถ้าการส่งผ่านค่าระหว่างฟังก์ชัน main() และฟังก์ชัน f() เป็นแบบ pass-by-value ผลลัพธ์ที่ได้จากคำสั่ง printf() คือค่าใด ?

ก. 0 0 0 ข. 1 0 3

ค. 1 3 0 ง. 0 1 3

20. การสร้างสภาพแวดล้อมเพื่อเอื้อต่อการพัฒนาซอฟต์แวร์ (software development environment) ปรากฏในภาษาใด ?

ก. UNIX shell programming ข. Script language

ค. Java Just-In-Time ง. Microsoft visual C++