Part A Programming Language Concepts

1 Introduction to programming languages

เนื้อหาใน Part A นี้จะอธิบายทฤษฎีเกี่ยวกับแนวคิดภาษาโปรแกรม ซึ่งพบได้ในภาษาโปรแกรมทั่วไป ประกอบไปด้วย 6 หัวข้อ ได้แก่

1. Introduction to programming languages
2. Name, scopes, and binding
3. Control flow
4. Data types
5. Subroutines and control abstraction
6. Data abstraction and object orientation

โดยหากมีการนิยามภาษาสำหรับเขียนโปรแกรมขึ้นใหม่ นิยามของภาษาก็จะประกอบไปด้วยส่วนต่าง ๆ ตามแนวคิดในหัวข้อต่าง ๆ ข้างต้น ดังนั้นหลาย ๆ ภาษาจึงความสอดคล้องและคล้ายคลึงกันอยู่

1.1 Programming Evolution

ปัจจุบันเป็นยุคที่เขียนโปรแกรมด้วยภาษาระดับสูง (high level languages) โดยภาษาที่เขียนจะใกล้เคียงกับภาษาธรรมชาติ (Natural language) กล่าวคือ เพื่อสั่งคอมพิวเตอร์ในทำการประมวลผลหนึ่ง ๆ ข้อความสั่ง (statement) จะถูกเขียนเป็นประโยคสั้น ๆ ซึ่งโดยทั่วไปจะใช้ภาษาอังกฤษ แต่ภาษาระดับสูงที่ใช้เขียนโปรแกรมนี้ไม่ใช่ภาษาที่คอมพิวเตอร์เข้าใจและประมวลผลได้ ดังนั้น จึงจำเป็นจะต้องถูกแปลงให้อยู่ในรูปแบบภาษาที่คอมพิวเตอร์เข้าใจได้ คือ ภาษาเครื่อง (Machine Language) ซึ่งอยู่ในรูปแบบของลำดับตัวเลขไบนารี ซึ่งในยุคเริ่มต้นนักคอมพิวเตอร์จะเขียนด้วยภาษานี้

ตัวอย่าง ชุดคำสั่ง สำหรับโปรเซสเซอร์สถาปัตยกรรม x86 ในรูปแบบฐาน 16 เช่น

```
55 89 e5 53   83 ec 04 83
00 00 39 c3   74 10 8d b6
75 f6 89 1c   24 e8 6e 00
```

เนื่องจากการที่โปรแกรมที่เขียนในรูปแบบชุดไบนารีมีความยากลำบากทั้งในการเขียนและการแก้จุดพร่อง (debug) โดยเฉพาะหากโปรแกรมนั้นมีขนาดที่ใหญ่มาก ด้วยปัญหานี้จึงเกิดวิวัฒนาการทางภาษาคอมพิวเตอร์ที่ช่วยให้สามารถเขียนโปรแกรมได้ง่ายขึ้น เป็นภาษาแอสเซมบลี (Assembly)

ตัวอย่างโปรแกรม สำหรับโปรเซสเซอร์สถาปัตยกรรม x86 เป็นดังนี้

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
subl  $4,%esp
```

Part A Programming Language Concepts

1 Introduction to programming languages

เนื้อหาใน Part A นี้จะอธิบายทฤษฎีเกี่ยวกับแนวคิดภาษาโปรแกรม ซึ่งพบได้ในภาษาโปรแกรมทั่วไป ประกอบไปด้วย 6 หัวข้อ ได้แก่

1. Introduction to programming languages
2. Name, scopes, and binding
3. Control flow
4. Data types
5. Subroutines and control abstraction
6. Data abstraction and object orientation

โดยหากมีการนิยามภาษาสำหรับเขียนโปรแกรมขึ้นใหม่ นิยามของภาษาก็จะประกอบไปด้วยส่วนต่าง ๆ ตามแนวคิดในหัวข้อต่าง ๆ ข้างต้น ดังนั้นหลาย ๆ ภาษาจึงความสอดคล้องและคล้ายคลึงกันอยู่

1.1 Programming Evolution

ปัจจุบันเป็นยุคที่เขียนโปรแกรมด้วยภาษาระดับสูง (high level languages) โดยภาษาที่เขียนจะใกล้เคียงกับภาษาธรรมชาติ (Natural language) กล่าวคือ เพื่อสั่งคอมพิวเตอร์ในทำการประมวลผลหนึ่ง ๆ ข้อความสั่ง (statement) จะถูกเขียนเป็นประโยคสั้น ๆ ซึ่งโดยทั่วไปจะใช้ภาษาอังกฤษ แต่ภาษาระดับสูงที่ใช้เขียนโปรแกรมนี้ไม่ใช่ภาษาที่คอมพิวเตอร์เข้าใจและประมวลผลได้ ดังนั้น จึงจำเป็นจะต้องถูกแปลงให้อยู่ในรูปแบบภาษาที่คอมพิวเตอร์เข้าใจได้ คือ ภาษาเครื่อง (Machine Language) ซึ่งอยู่ในรูปแบบของลำดับตัวเลขไบนารี ซึ่งในยุคเริ่มต้นนักคอมพิวเตอร์จะเขียนด้วยภาษานี้

ตัวอย่าง ชุดคำสั่ง สำหรับโปรเซสเซอร์สถาปัตยกรรม x86 ในรูปแบบฐาน 16 เช่น

```
55 89 e5 53   83 ec 04 83
00 00 39 c3   74 10 8d b6
75 f6 89 1c   24 e8 6e 00
```

เนื่องจากการที่โปรแกรมที่เขียนในรูปแบบชุดไบนารีมีความยากลำบากทั้งในการเขียนและการแก้จุดพร่อง (debug) โดยเฉพาะหากโปรแกรมนั้นมีขนาดที่ใหญ่มาก ด้วยปัญหานี้จึงเกิดวิวัฒนาการทางภาษาคอมพิวเตอร์ที่ช่วยให้สามารถเขียนโปรแกรมได้ง่ายขึ้น เป็นภาษาแอสเซมบลี (Assembly)

ตัวอย่างโปรแกรม สำหรับโปรเซสเซอร์สถาปัตยกรรม x86 เป็นดังนี้

```
pushl %ebp
movl  %esp,%ebp
pushl %ebx
subl  $4,%esp
```

ภาษาแอสเซมบลี ภาษาเฉพาะเครื่อง (Machine Specific) ซึ่งจะนิยามขึ้นตามแต่ละสถาปัตยกรรมคอมพิวเตอร์ ดังนั้น สถาปัตยกรรมของหน่วยประมวลผลที่แตกต่างกัน จะใช้ชุดคำสั่งภาษาแอสเซมบลีที่แตกต่างกันไปโดยเฉพาะและไม่สามารถนำไปประมวลผลข้ามสถาปัตยกรรมได้ โดยภาษาแอสเซมบลีมีลักษณะเป็นอักษรย่อช่วยจำ (mnemonic abbreviations) โดยจะใช้ตัวอักษรย่อเป็นสัญลักษณ์แทนชุดคำสั่งไบนารีต่าง ๆ ของภาษาเครื่อง ทำให้สามารถจดจำและเข้าใจได้ง่ายขึ้น โดยคำสั่งที่ปรากฏในตัวอย่างข้างต้นเช่น pushl movl และ subl เป็นต้น

จากตัวอย่างเป็นการเขียนภาษา assembly ที่เป็น machine specific ซึ่งเป็นลักษณะที่เขียน code เพื่อรันบน computer architecture แบบหนึ่งจึงต้องมี assembly language เฉพาะสำหรับ architecture แบบนั้น แต่ assembly เป็นภาษาที่เรียกว่า mnemonic abbreviations หมายความว่า ภาษาเริ่มมีสัญลักษณ์เพื่อช่วยให้เราจำได้ว่าคำสั่งต่างๆที่ต้องการสั่งให้เครื่องคอมพิวเตอร์ทำงานคืออะไร จากตัวอย่าง เราจะเห็นคำสั่งพวก pushl , movl , subl เป็นต้น

วิวัฒนาการลำดับต่อมาได้มีภาษาโปรแกรมเกิดขึ้นอีกมากมายนับร้อยภาษา ซึ่งมีลักษณะใกล้เคียงกับภาษาธรรมชาติมากขึ้น เช่น Fortran, Lisp, Prolog, Pascal, C, C++ เป็นต้น (สามารถดูรายการภาษาโปรแกรมต่างได้ตามลิงค์ https://en.wikipedia.org/wiki/List_of_programming_languages) โดยโปรแกรมที่เขียนขึ้นในภาษาเหล่านี้จะถูกนำไปประมวลผลโดยคอมพิวเตอร์ด้วยการแปลงในอยู่ในรูปภาษาเครื่องด้วยวิธีการต่าง ๆ กันไปก่อนการประมวลผลต่อไป

1.1.1 Why So many?

สาเหตุที่ภาษาโปรแกรมมีจำนวนมากมายหลากหลายมีดังนี้

1.1.1.1 การวิวัฒนาการ (Evolution)

ภาษาโปรแกรมที่มีอยู่นับร้อยนั้นถูกนิยามขึ้นในช่วงเวลาต่าง ๆ กันไป บางภาษาก็นิยามขึ้นใหม่ บางภาษาก็พัฒนาปรับปรุงจากภาษาที่มีอยู่เดิม ตัวอย่างการวิวัฒนาการอย่างเช่น ภาษาในรุ่นเก่า เช่น Fortran, COBOL, Basic จะมีคำสั่ง goto ซึ่งเป็นการสั่งให้ประมวลผลยังบรรทัดที่มีชื่อที่กำหนดไว้ได้

ตัวอย่างของการเขียนโปรแกรมโดยใช้คำสั่ง goto เช่น

```
B: …
…
goto A
…
A: …
goto B
```

จากตัวอย่างมีการระบุป้ายชื่อ (label) กำกับบรรทัด ได้แก่ A และ B สำหรับใช้กับคำสั่ง goto ให้กระโดดข้ามไปประมวลผลยังบรรทัดเหล่านั้นได้ โดยป้ายชื่อจะนิยามไว้กับบรรทัดโดยขึ้นต้นด้วยข้อความปิดด้วยเครื่องหมาย colon ซึ่งเป็นคำสั่งสำหรับ control flow ในการเขียนโปรแกรม ด้วยการกระโดดไปมาของคำสั่งเมื่อโปรแกรมมีความซับซ้อนโปรแกรมอาจทำให้โปรแกรมที่ถูกเขียนขึ้นมีลักษณะที่ยุ่งเหยิง เรียกว่า spaghetti code ทำให้เกิดปัญหาในการทำความเข้าใจและแก้ไขข้อบกพร่องของโปรแกรมได้จากความไม่เป็นระเบียบของโปรแกรม

ด้วยปัญหาของความยากลำบากจากการใช้คำสั่ง goto ภาษารุ่นถัดมาจึงเปลี่ยนเป็นยุคของ structured programming โดยภาษาจะมีโครงสร้างชัดเจนและเป็นระเบียบมากขึ้น โดยคำสั่งกลุ่ม control flow จะมีโครงสร้างเป็นลักษณะ block บรรจุคำสั่งอยู่ภายใน เช่น คำสั่ง while เป็นการวนลูปเพื่อประมวลผลชุดคำสั่งภายในขอบเขตของ block

จากนั้น จึงเป็นยุคภาษาโปรแกรมเชิงวัตถุ (Object-Oriented: OO) ซึ่งพัฒนาขึ้นเพื่อให้การเขียนโปรแกรมมีการการแก้ปัญหาที่สอดคล้องกับวิธีธรรมชาติของมนุษย์ ตัวอย่างการแก้ปัญหาเช่น เมื่อพรมในรถเกิดเปียกเพราะน้ำท่วม หากต้องการทำให้พรมแห้ง การแก้ปัญหาคือนำรถไปที่อู่ให้ช่างดำเนินการให้ กล่าวคือ หากต้องการแก้ปัญหาใด ๆ การแก้ปัญหาจะทำโดยการหาว่ามีใครที่สามารถแก้ปัญหาให้ได้บ้าง โดยอาจไม่ต้องทราบว่าการแก้ปัญหานั้นมีรายละเอียดอย่างไรบ้าง เช่นการทำให้พรมแห้งเราไม่ต้องรู้ว่าใช้วิธีการอย่างไรแต่รู้ว่าช่างสามารถทำให้พรมแห้งได้ เทียบกับการเปิดหาวิธีการผ่านระบบอินเทอร์เน็ตเพื่อหาขั้นตอนการตากพรมว่ามีรายละเอียดอย่างไรซึ่งเป็นลักษณะการแก้ปัญหาแบบ procedural ซึ่งเป็นวิธีการแก้ปัญหาของ structured programming เช่นในการเขียนโปรแกรมภาษาซี โปรแกรมจะเริ่มจากฟังก์ชัน main จากนั้นจึงเรียกฟังก์ชันต่าง ๆ และฟังก์ชันเหล่านั้นอาจมีการเรียกฟังก์ชันอื่น ๆ ต่อไปอีก โดยในแต่ละฟังก์ชันจะมีรายละเอียดขั้นตอนของการแก้ปัญหาต่างจากการแก้ปัญหาเชิงวัตถุที่จะต้องออกแบบคลาส (class) ซึ่งอธิบายคุณลักษณะของวัตถุ (object) ว่ามีวัตถุใดที่มีความสามารถใดบ้างที่ต้องทำงานร่วมกันเพื่อแก้ไขปัญหาให้สำเร็จได้ จากนั้นจึงลงรายละเอียดการกระทำของวัตถุเหล่านั้นว่าแก้ปัญหาได้อย่างไรซึ่งสะท้อนวิธีการแก้ปัญหาของมนุษย์ได้ดีขึ้น จึงเป็นการก้าวเข้าสู่ยุคการเขียนโปรแกรมเชิงวัตถุในปัจจุบัน

1.1.1.2 จุดประสงค์เฉพาะ (Special purposes)

เนื่องจากปัญหาที่ต้องการแก้มีความหลากหลาย บางครั้งภาษาทั่วไป (general purpose) ไม่สามารถตอบสนองความต้องการในการแก้ปัญหาได้ทั้งหมด จึงได้มีภาษาสำหรับเขียนโปรแกรมเฉพาะสำหรับการใช้งานด้านต่าง ๆ โดยเฉพาะ

ตัวอย่างเช่น ภาษา C เหมาะสำหรับการเขียนโปรแกรมระบบระดับต่ำ (low level system programming) จำพวก Network, OS, Driver ต่าง ๆ เป็นต้น ภาษา Lisp เป็นภาษาที่เด่นในเรื่องของการทำงานกับลิสต์ (list manipulation) โดยชื่อภาษาย่อมาจาก List processing ภาษา Prolog เป็นภาษาที่นิยมใช้ในงานด้านปัญญาประดิษฐ์ (Artificial Intelligence: AI)

1.1.2 Why a few are widely used?

แม้ว่าภาษาโปรแกรมมีจำนวนมากมาย แต่มีเพียงบางภาษาเท่านั้นที่เป็นที่นิยม สาเหตุเนื่องมาจากปัจจัยต่างเหล่านี้

1.1.2.1 Expressive power

ความสามารถที่พิเศษกว่าภาษาอื่นบางอย่างทำให้ได้รับความนิยม อย่างเช่น garbage collection เป็นฟังก์ชันที่คอยจัดการคืนพื้นที่หน่วยความจำหลักที่เก็บข้อมูลที่ไม่ถูกใช้ในโปรแกรมแล้วให้สามารถนำมาใช้ใหม่ได้ สำหรับในภาษาที่ไม่มี garbage collection เช่น ภาษา C เวลาเขียนโปรแกรมจะต้องทำการจัดการหน่วยความจำหลักเอง ตั้งแต่การจองพื้นที่ ใช้งานและคืนพื้นที่หน่วยความจำ ซึ่งทำให้การเขียนโปรแกรมทำได้อย่างยากลำบากเพราะหากจัดการได้ไม่ดีพออาจจะทำให้โปรแกรมทำงานล้มเหลวได้ ดังนั้นการมี garbage collection จึงช่วยลดภาระในการเขียนโปรแกรมให้กับโปรแกรมเมอร์ทำให้ภาษานั้น ๆ ได้รับความนิยม

1.1.2.2 Ease of use for novice

ความง่ายของภาษาสำหรับผู้เริ่มต้นเป็นปัจจัยหนึ่งที่ได้รับความนิยมเพื่อเป็นภาษาสำหรับเริ่มฝึกการเขียนโปรแกรมเพื่อพัฒนาไปภาษาที่ยากกว่า เช่น Python กับ Pascal ง่ายกว่า Java จึงถูกใจผู้เรียนมากกว่า แต่ Java ก็ยังง่ายกว่า C++

1.1.2.3 Ease of implementation

เมื่อพิจารณาในการนิยามภาษาโปรแกรมใด ๆ ขึ้น จะมีข้อกำหนดต่าง ๆ ของภาษา เช่นการตั้งชื่อตัวแปรอย่างไร ชนิดข้อมูลเป็นแบบใด เป็นต้น จากนั้นจะมีผู้นำข้อกำหนดเหล่านี้ไปสร้างตัวแปลภาษาสำหรับแปลและประมวลผลโค้ดที่เขียนขึ้นตามข้อกำหนดได้ ซึ่งภาษาหนึ่ง ๆ อาจมีความหลากหลายของตัวแปลภาษา เช่นมีหลายผู้ผลิตอย่าง ภาษาจาวาสคริปต์ หรืออาจทำงานได้ทั้งแบบ compiler หรือ interpreter ตัวแปลภาษาเหล่านี้ เรียกได้ว่าเป็น implementation ของภาษา

ความนิยมอาจเกิดจากการจัดหา implementation ได้ง่าย เช่น กลุ่มภาษาที่สามารถดาวน์โหลดไปใช้งานได้ฟรี เช่น ภาษา Pascal, Java Python เป็นต้น

สำหรับบางภาษาก็อาจได้รับความนิยมจากตัวแปลภาษาที่ใช้ทรัพยากรระบบในการประมวลผลน้อย

1.1.2.4 Standardization

การที่ implementation ของภาษานั้นมีข้อกำหนดมาตรฐานเกี่ยวกับไลบรารี (libraries) พื้นฐานมาด้วย จะช่วยให้การเขียนโปรแกรมทำได้ง่าย โยกย้ายได้สะดวกและมีแบบแผน เช่น การเขียนโปรแกรมภาษา Java จะมี Java System Class จำนวนมากให้เลือกใช้งาน โดยที่ไม่ต้องเขียนขึ้นเอง

ต่างจากภาษาที่ไม่ได้เตรียมข้อกำหนดไลบรารีพื้นฐานให้ ภาระเหล่านี้ก็จะตกยังผู้ผลิตตัวแปลภาษาที่ต้องพัฒนาไลบรารีต่าง ๆ เพื่อความสะดวกต่อโปรแกรมเมอร์ขึ้นเอง ซึ่งอาจทำให้สูญเสียความเป็นอันหนึ่งอันเดียวกันของภาษาได้ เช่น การจะใช้งานโปรแกรมที่เขียนขึ้นต้องใช้ตัวแปลภาษาเฉพาะของบริษัทใดบริษัทหนึ่งเท่านั้น

1.1.2.5 Open source

การเปิดเผยซอร์ซโค้ดของ compiler หรือ interpreter ก็มีส่วนต่อความนิยม เช่นกัน ตัวอย่างความนิยมใน open source เช่น ระบบปฏิบัติการ UNIX ที่ถูกพัฒนาขึ้นด้วยภาษา C และถูกแจกจ่ายไปยังหน่วยงานต่าง ๆ พร้อมกับ compiler ภาษา C หลังจากนั้น หน่วยงาน บริษัท มหาวิทยาลัย ก็ได้พัฒนา UNIX เวอร์ชันของตนเองขึ้น ทำให้มี UNIX ที่นำไปปฏิบัติการบนแพลตฟอร์มต่าง ๆ ได้

1.1.2.6 Excellent compilers and tools

ภาษาจะถูกเลือกใช้จากความยอดเยี่ยมในด้านต่าง ๆ ของตัวแปลภาษา เช่น ตัวแปลภาษาที่สามารถสร้างโปรแกรมที่ทำงานได้รวดเร็ว อย่างภาษา Fortran เป็นต้น

ภาษาเครื่องมือพัฒนาโปรแกรมที่สามารถช่วยให้เขียนโปรแกรมได้ง่ายและรวดเร็วขึ้น เช่น การตรวจสอบความถูกต้องของโค้ด เครื่องมือช่วยเขียนโค้ด (code assistant) รวมไปถึงเครื่องมือช่วยในการแก้ไขข้อบกพร่อง (debugger) ก็มีส่วนให้ภาษามีความนิยมเพราะความสะดวกในการพัฒนา

1.1.2.7 Economics, patronage, and inertia

ปัจจัยจากการสนับสนุนจากองค์กรทำให้บางภาษามีการใช้งานแพร่หลาย เช่น ในอดีตกระทรวงกลาโหมสหรัฐอเมริกา (US Department of Defense: DoD) ได้ส่งเสริมให้ใช้ภาษาโปรแกรมจำนวนหนึ่งในการพัฒนาโปรแกรมให้กับหน่วยงานภาครัฐ ต่อมาจึงได้กำหนดภาษาเฉพาะเจาะจงสำหรับงานด้านต่าง ๆ เริ่มด้วยกำหนดให้ใช้ภาษา COBOL สำหรับการพัฒนาโปรแกรมด้านการประมวลผลข้อมูล (data processing) คำว่า COBOL ย่อมาจาก Common Business Oriented Language เป็นภาษาที่ใช้สำหรับการประมวลผลข้อมูลปริมาณมากบนคอมพิวเตอร์เมนเฟรม อย่างการออกรายงานต่าง ๆ ของ

ธนาคารที่มีข้อมูลขนาดใหญ่ เช่น ข้อมูลลูกค้าและพนักงาน ซึ่งถือเป็น Common Business Function ต่อมาก็มีภาษา Ada ซึ่งวิวัฒนาการมาจากภาษา Pascal ซึ่งยังคงมีการใช้งานอยู่ และถูกปรับปรุงภาษาอย่างต่อเนื่องจนกระทั่งมีความสามารถในการเขียนโปรแกรมเชิงวัตถุ

ภาษา C# ก็ยังได้รับการสนับสนุนจาก Microsoft DoD และ Objective-C ก็ได้รับการสนับสนุนจาก Apple เช่นกัน

นอกจากนี้ปัจจัยทางเศรษศาสตร์อย่างเช่นความคุ้มค่าในการแทนที่ด้วยภาษาใหม่ กรณีที่ชัดเจนที่สุด คือ ภาษา COBOL ที่ถูกใช้ในการเขียนโปรแกรมด้านการธนาคารและปัจจุบันยังคงใช้งานอยู่นั้น ในอดีตระบบคอมพิวเตอร์เป็นระบบที่มีราคาสูงและโดยเฉพาะการประมวลผลข้อมูลจำนวนมากจะต้องใช้คอมพิวเตอร์ขนาดเมนเฟรมยิ่งมีต้นทุนที่สูงมาก และเมื่อโปรแกรมจำนวนมากที่เขียนด้วยภาษา COBOL ยังคงทำงานได้อยู่ การเปลี่ยนแปลงจึงไม่มีความจำเป็นเพราะจะเป็นการเพิ่มต้นทุนระบบยิ่งขึ้นโดยไม่จำเป็น อีกตัวอย่างหนึ่งคือโปรแกรมซื้อขายหลักทรัพย์ของตลาดหลักทรัพย์ที่ถูกเขียนขึ้นมาแล้วเป็นระยะเวลาหลายสิบปีก็ยังคงถูกใช้งานอยู่ แต่สำหรับฟังก์ชันใหม่ที่พัฒนาขึ้นจึงใช้เทคโนโลยีที่สมัยขึ้นแทน

1.2 Classification of programming languages

ภาษาโปรแกรมสามารถแบ่งออกเป็น 2 ประเภท คือ Imperative กับ Declarative

1.2.1 Imperative

Imperative language หรือ ภาษาเชิงคำสั่ง เป็นภาษาที่เขียนโปรแกรมในลักษณะสั่งการให้คอมพิวเตอร์ประมวลผลอย่างเป็นลำดับขั้นตอน โดยลำดับการประมวลผลคำสั่งมีผลต่อผลลัพธ์การทำงาน ตัวอย่างเช่น คำสั่งจัดการตัวแปรที่อ้างถึงค่าที่บันทึกไว้ในหน่วยความจำ ซึ่งตัวแปรหนึ่งอาจถูกกำหนดค่าหรือดึงค่าหลายครั้ง ค่าที่ดึงมาได้ของตัวแปรในแต่ละครั้งอาจจะเปลี่ยนไป ลำดับคำสั่งในการดึงและกำหนดค่าตัวแปรจึงมีความสำคัญ

ต่อไปนี้เป็นตัวอย่างการเขียนโปรแกรมเชิงคำสั่งเพื่อหาค่าหารร่วมมากของจำนวนเต็ม 2 จำนวน

```
int gcd(int a, int b){
    while(a != b) {
            if(a > b) a = a - b;
            else b = b - a;
    }
    return a;
}
```

จากตัวอย่างโค้ด เป็นชุดคำสั่งสำหรับหาค่าหารร่วมมาก โดยมีลำดับขั้นตอน คือ ตรวจสอบว่าค่าตัวแปร a และ b มีค่าไม่เท่ากันหรือไม่ หากไม่เท่าให้วนลูปเพื่อประมวลผลลำดับคำสั่งภายในบล็อกจนกว่าจะค่าตัวแปรจะเท่ากัน โดยคำสั่งภายในบล็อกจะเป็นค่าเลือกกำหนดค่าตัวแปร a หรือ b ใหม่ โดยแต่ละรอบในการวนลูปจะมีค่าตัวแปรใดตัวแปรหนึ่งถูกกำหนดค่าใหม่ ทำให้สถานะในคำนวณหรือประมวลผลเปลี่ยนไปในแต่ละรอบ

ภาษาเชิงคำสั่งมีการแบ่งย่อยตามวิวัฒนาการ ดังนี้

1.2.1.1 Von Neumann

Von Neumann ตั้งตามชื่อของนักคณิตศาสตร์เพื่อเป็นเกียรติแก่ท่าน สำหรับภาษาโปรแกรมเชิงคำสั่งในยุคแรก ๆ เช่น Fortran, Pascal, Basic, C เป็นต้น จะถูกจัดอยู่ในประเภทนี้

Von Neumann เป็นนักคอมพิวเตอร์ที่เสนอแนวคิดสถาปัตยกรรม Von Neumann โดยจะจัดให้โปรแกรม (program) กับข้อมูล (data) เป็นสิ่งเดียวกัน คือ จัดให้โปรแกรมเป็นข้อมูลประเภทหนึ่ง เพราะในขณะที่โปรแกรมทำงานตัวโปรแกรมเองรวมไปถึงข้อมูลที่ใช้จะถูกจัดเก็บไว้ในหน่วยความจำร่วมกัน ซึ่งแตกต่างจากสถาปัตยกรรมในยุคก่อนหน้าที่แยกหน่วยความจำสำหรับโปรแกรมและข้อมูลแยกออกจากกัน

1.2.1.2 Scripting – Interpreted

ภาษาสคริปต์ เป็นภาษาโปรแกรมที่มีรูปแบบการเขียนอย่างสั้นและง่ายในลักษณะการเรียงร้อยคำสั่งที่มีอยู่แล้วอย่างไลบรารีต่าง ๆ เข้าด้วยกัน ที่มาของของประเภท scripting เริ่มจากยุคแรกเกิดมี shell script ซึ่งสำหรับเขียนลำดับคำสั่ง shell ของระบบปฏิบัติการ UNIX รวมเป็นไฟล์เดียวและสามารถสั่งประมวลผลไฟล์นี้ได้ โดยให้ผลเหมือนกับการป้อนคำสั่งไปยัง shell terminal ทีละคำสั่ง ตัวอย่าง shell script เช่น

```
echo "hello"
date
```

คำสั่งข้างต้นในแต่ละบรรทัด คือ คำสั่ง shell 1 คำสั่ง โดยเป็นการสั่งพิมพ์คำว่า hello ตามด้วยวันที่และเวลา

โดยภาษาสคริปต์ที่เขียนขึ้นนี้จะสามารถนำไปประมวลผลในสภาพแวดล้อมโดยเฉพาะ อย่าง shell script ในตัวอย่างข้างต้นก็สามารถทำงานได้บน UNIX shell เท่านั้น หรือภาษาสคริปต์อื่น ๆ เช่น ภาษา JavaScript ที่สามารถเขียนโค้ดฝังไว้กับโค้ด HTML ก็จะสามารถนำไปประมวลผ่าน Web Browser ซึ่งมีอินเทอร์พรีเตอร์สำหรับประมวลผลภาษา JavaScript โดยเฉพาะ หรือภาษา PHP ก็จะถูกประมวลผลโดย PHP interpreter ซึ่งถูกติดตั้งไว้บน web server เป็นต้น

โดยสรุปแล้ว ภาษาสคริปต์ส่วนใหญ่จะเขียนได้ง่าย และทำงานบนสภาพแวดล้อมประมวลผลพิเศษโดยเฉพาะของตัวภาษาเองในลักษณะของ อินเทอร์พรีเตอร์ (interpreter) โดยภาษาสคริปต์จะรวมไปถึง dynamic general-purpose language เช่น ภาษา Python การเขียนโปรแกรมด้วยภาษานี้ก็จะถูกเรียกว่าเขียน Python script เนื่องจากมีรูปแบบคำสั่งที่ง่ายและถูกประมวลผลด้วยอินเทอร์พรีเตอร์

1.2.1.3 Object-Oriented (OO)

ภาษาโปรแกรมเชิงวัตถุมีวิวัฒนาการมาจากภาษาเชิงคำสั่ง เช่น ภาษา C++ และภาษา Java พัฒนามาจากภาษา C จึงยังคงมีส่วนของโค้ดที่จะมีลักษณะของ imperative อยู่ได้แก่คำสั่งที่บรรจุอยู่ภายในแต่ละเมธอดของคลาส เพียงแต่ OO จะมีส่วนหลักของภาษาที่ว่าด้วยคลาสหรืออ็อบเจกต์ ซึ่งบรรจุเมธอดและข้อมูลไว้ภายใน โดยความสัมพันธ์ระหว่างคลาสจะเป็นแบบ semi-independent คือ มีความเป็นอิสระระหว่างกัน แต่ยังมีความสัมพันธ์ในการเรียกใช้ซึ่งกันและกันอยู่

1.2.2 Declarative

Declarative language หรือ ภาษาเชิงประกาศ เป็นภาษาที่ไม่ได้กำหนดลำดับขั้นตอนของการประมวลผลเพื่อหาคำตอบ แต่จะกำหนดคุณสมบัติของคำตอบของปัญหา

1.2.2.1 Functional

ภาษาประเภทนี้ เช่น Lisp, Scheme, ML, Haskell เป็นต้น ตัวอย่างโค้ดเพื่อหาค่าหารร่วมมากด้วยภาษา Ocaml ซึ่งเป็นภาษาตระกูลของ ML เป็นดังนี้

```
let rec gcd a b =
   if a = b then a
   else if a > b then gcd b (a - b)
          else gcd a (b - a)
```

จากตัวอย่างเป็นการนิยามคุณสมบัติในการหาค่าหารร่วมมากเป็นฟังก์ชันชื่อว่า gcd โดยจะสังเกตเห็นว่าในนิยามมีการใช้ฟังก์ชันแบบรีเคอร์ซีฟ (recursive) ด้วย

1.2.2.2 Logic

ภาษา Prolog เป็นตัวอย่างภาษาประเภทนี้ โดยจะไม่กำหนดลำดับขั้นของการทำงานแต่บอกคุณสมบัติของการแก้ปัญหาในรูปแบบของกฎ (rule) ตัวอย่างการเขียนกฎในการหา ห.ร.ม. เป็น ดังนี้

```
gcd(A, B, G) :- A = B, G = A.
gcd(A, B, G) :- A > B, C is A-B, gcd(C, B, G).
gcd(A, B, G) :- B > A, C is B-A, gcd(C, A, G).
```

จากตัวอย่างนิยามกฎชื่อ gcd โดยกำหนดให้ G เป็นผลลัพธ์ของการหาค่า ห.ร.ม. ของ A และ B โดยมีนิยามกฎจำนวน 3 กฎ ตามความสัมพันธ์ระหว่าง A กับ B และวิธีการในการหาผลลัพธ์ในแต่ละแบบของความสัมพันธ์ โดยเครื่องหมาย :- จะแสดงความเป็นเหตุเป็นผลต่อกัน โดยค่าฟังก์ชันทางด้านซ้ายของเครื่องหมายจะเป็นจริงถ้า กฎที่นิยามไว้ทางด้านขวาของเครื่องหมาย ซึ่งแต่ละกฎคั่นไว้ด้วยเครื่องหมาย , ซึ่งมีความหมายเป็น AND หมายถึงทุกกฎต้องเป็นจริงทั้งหมด

เมื่อสอบถามผลลัพธ์ ห.ร.ม. โปรแกรม Prolog จะประมวลผลโดยหาค่าของ G ที่เป็นไปได้ซึ่งเมื่อแทนค่าลงไปในกฎใดกฎหนึ่งที่ถูกนิยามไว้เรียกว่า solution space แล้วให้ผลลัพธ์ที่เป็นจริงได้ค่านั้นก็จะเป็นคำตอบของปัญหา

สิ่งสำคัญหนึ่งในการเรียน programming language คือ ควรจะเรียนรู้ทั้งภาษาประเภท imperative และ declarative หรือเรียกได้ว่าเรียนรู้แบบ alternative programming style

1.3 Why study programming languages?

การเรียนภาษาโปรแกรมจะช่วยให้สามารถเลือกใช้ภาษาที่เหมาะสมกับงานได้ และเมื่อเข้าใจในแนวคิดพื้นฐานของภาษาโปรแกรมจะช่วยให้สามารถเรียนรู้ภาษาโปรแกรมใหม่ ๆ ได้โดยง่าย เนื่องจากหลาย ๆ ภาษาก็จะมีความคล้ายคลึงกันอยู่ เช่น ระหว่างภาษา Java C# และ C++ เป็นต้น และมีการใช้แนวคิดพื้นฐานเดียวกัน เช่น ชนิดข้อมูล และ control flow เป็นต้น

การเรียนภาษาโปรแกรมจะช่วยให้เข้าใจทางเลือกการในการเขียนโปรแกรม ซึ่งบางครั้งการเขียนเพื่อหาผลลัพธ์หนึ่ง ๆ ในแง่มุมของตรรกะอาจสามารถเขียนโปรแกรมได้หลายวิธี แต่ในเบื้องลึกของการทำงานของภาษาโปรแกรมแต่ละวิธีจะมีการใช้ทรัพยากรที่แตกต่างกัน อย่างเช่น คำสั่ง switch case สามารถเขียนให้อยู่ในรูปแบบ nested if ได้ ซึ่งแต่ละวิธีใช้ทรัพยากรเพื่อการประมวลผลมากน้อยแตกต่างกันไปตามจำนวนกรณีตัวเลือกในการประมวลผล ซึ่งเมื่อทราบถึงรายละเอียดการทำงานของภาษาจะช่วยให้เลือกวิธีการที่เหมาะสมที่สุดได้

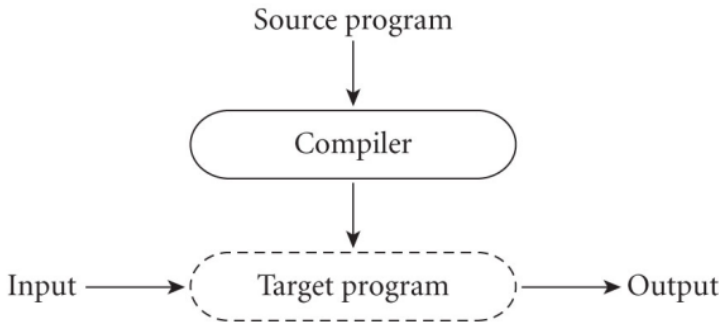
กรณีที่โปรแกรมเกิดข้อผิดพลาดขึ้น การเข้าใจในภาษาโปรแกรมจะช่วยให้ง่ายในค้นหาต้นเหตุของข้อผิดพลาดนั้น ๆ ได้

1.4 Program Translation: Compiler

ภาพที่ 1.1 แผนภาพแสดงการทำงานของ Compiler

คำบรรยายภาพ: ขั้นตอนการทำงานของ Compiler

1. Source Program เป็นข้อมูลนำเข้าสำหรับประมวลผลโดย Compiler
2. Compiler วิเคราะห์และแปล Source Program ได้ผลลัพธ์เป็น Target Program
3. Input เป็นข้อมูลนำเข้าของ Target Program
4. Target Program ทำการประมวลผลข้อมูลที่นำเข้าและให้ผลลัพธ์เป็น Output



โปรแกรมที่ถูกเขียนขึ้นด้วยภาษาโปรแกรมจะได้ผลลัพธ์เป็น Source Program ภาษาที่ทำงานแบบ Compiler นั้น Source Program จะยังไม่สามารถนำไปสั่งให้คอมพิวเตอร์ให้ประมวลผลได้ จะต้องถูกแปลให้อยู่ในรูปแบบที่ประมวลผลได้โดย Compiler ซึ่งให้ผลลัพธ์เป็น Target Program ที่สามารถนำไปสั่งให้คอมพิวเตอร์ทำการประมวลผลได้ โดยเมื่อสั่งให้โปรแกรมทำงานก็จะรับ Input และประมวลผลได้ Output ต่อไป

การทำงานแบบ Compiler จะแปล Source Program ให้เป็นภาษาเครื่องเพียงครั้งเดียว Target Program สามารถนำไปทำงานได้โดยไม่ต้องทำการแปลใหม่อีกยกเว้นโปรแกรมมีการเปลี่ยนแปลงหรือแก้ไข

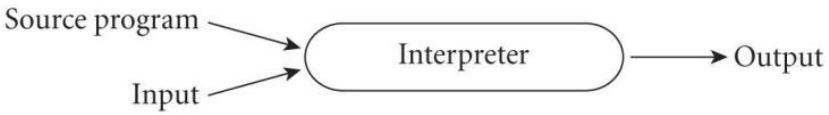
ข้อดีของการแปลแบบ Compiler คือ ในด้านสมรรถนะของโปรแกรม เพราะการตัดสินใจดำเนินการบางอย่างสามารถดำเนินการได้ตั้งแต่ในขั้นตอนของการคอมไพล์ ทำให้ขั้นตอนการทำงานในขณะประมวลผลโปรแกรมลดลง โปรแกรมจึงทำงานได้เร็วขึ้น เช่น การกำหนดที่อยู่หน่วยความจำ (memory address) ของตัวแปร ที่สามารถกำหนดตั้งแต่การคอมไพล์และฝังไว้ใน Target Program ได้เลย ดังนั้นเมื่อ ถึงเวลาโปรแกรมทำงานก็จะสามารถระบุตำแหน่งของตัวแปรได้ทันที ไม่ต้องมีการพิจารณาหาตำแหน่งของตัวแปรอีก

1.5 Program Translation: Interpreter

ภาพที่ 1.2 แผนภาพแสดงขั้นตอนการทำงานของอินเทอร์พรีเตอร์

คำบรรยายภาพ: ขั้นตอนการทำงานของ Interpreter

1. ข้อมูลนำเข้าประกอบด้วย Source Program และ Input ซึ่งจะส่งให้ Interpreter ดำเนินการต่อ
2. Interpreter ทำการแปลภาษาและกระทำการตามคำสั่งกับข้อมูลนำเข้า
3. Interpreter ให้ผลลัพธ์เป็น Output



การแปลภาษาโปรแกรมด้วยอินเทอร์พรีเตอร์ (interpreter) จะแปลและกระทำการ (execute) คำสั่งโดยตรงจาก Source program กล่าวคือ อินเทอร์พรีเตอร์จะรับ Source program และ Input เป็นข้อมูลนำเข้า จากนั้น อินเทอร์พรีเตอร์จะวิเคราะห์ Source Program ว่าเขียนได้ถูกต้องตามหลักภาษาหรือไม่ หากพบว่าถูกต้องแล้วจะทำการแปลเป็นภาษาเครื่องและกระทำคำสั่งนั้นทันที จนได้ Output จากการประมวลผลเป็นผลลัพธ์ ซึ่งต่างจาก Compiler คือ ไม่ได้แบ่งออกเป็น 2 ขั้นตอน อย่างแปลเป็นภาษาเครื่องก่อนได้เป็น object code ก่อน จึงนำไปกระทำการซ้ำ ๆ ได้โดยไม่ต้องแปลเป็นภาษาเครื่องอีก ดังนั้น เมื่อมีการสั่งประมวลผลโปรแกรมด้วยอินเทอร์พรีเตอร์ทุกครั้ง Source Program จะต้องถูกแปลเป็นภาษาเครื่องทุก ๆ ครั้ง แม้ว่าโปรแกรมจะไม่มีการแก้ไขเปลี่ยนใด ๆ เลย

ในแต่ละขั้นตอนของการแปลและกระทำการของอินเทอร์พรีเตอร์ไม่ได้ทำกับทั้ง Source Program แต่จะแปลและกระทำการทีละข้อความสั่ง (Statement) หรือดำเนินการแบบบรรทัดต่อบรรทัด

สำหรับเบื้องหลังการทำงานของอินเทอร์พรีเตอร์นั้น การแปล Source Program ไม่ได้แปลเป็นภาษาเครื่องโดยตรง แต่จะได้เตรียมฟังก์ชันต่าง ๆ ในรูปของภาษาเครื่องไว้ก่อนแล้ว การแปลจะทำเพื่อเรียกฟังก์ชันเหล่านี้แทน ตัวอย่างเช่น การแปลข้อความสั่งบวกเลข 2 จำนวน เช่น a + b แทนที่อินเทอร์พรีเตอร์จะแปลคำสั่งเป็นชุดคำสั่งภาษาเครื่องสำหรับการคำนวณบวกเลข จะทำการเรียกฟังก์ชัน add โดยส่งค่าตัวแปร a และ b ซึ่งฟังก์ชันได้ถูกเตรียมเป็นภาษาเครื่องไว้เรียบร้อยก่อนแล้ว

ข้อเด่นของอินเทอร์พรีเตอร์ คือ สามารถตรวจสอบและแก้ไขข้อบกพร่องได้ง่าย เพราะการกระทำการจะทำกับ Source Program โดยตรงทำให้เวลาเกิดข้อผิดพลาดจะสามารถระบุตำแหน่งที่เกิดปัญหาใน Source Program ได้โดยตรง ซึ่งได้เปรียบกว่า Compiler ที่กระทำการกับโปรแกรมที่เปลี่ยนรูปไปแล้ว

ข้อด้อยของอินเทอร์พรีเตอร์ คือ การประมวลผลต่าง ๆ จะช้ากว่าใช้ Compiler เพราะการแปลและกระทำการจะถูกดำเนินการในขณะ run-time ตัวอย่างเช่น บางข้อความสั่งจะถูกแปลและกระทำการซ้ำหลาย ๆ ครั้งทั้งที่เป็นข้อความสั่งเดียวกัน อย่างข้อความสั่งในลูป กล่าวคือ อินเทอร์พรีเตอร์จะเสียทรัพยากรระบบเพิ่มในส่วนของการแปลที่ทำทุกครั้งที่กระทำการคำสั่งใด ต่างจาก compiler ที่ก่อนการกระทำการเพียงครั้งเดียว

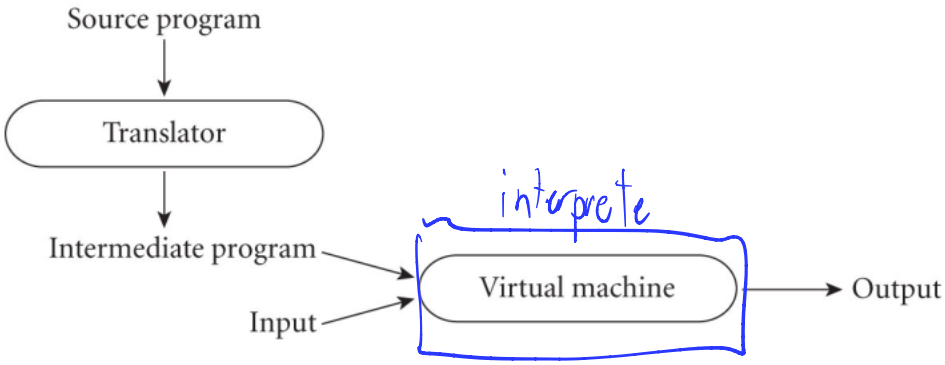
1.6 Combination of Compiler and Interpreter

เป็นการผสมผสานข้อดีของ compiler กับ interpreter เข้าด้วยกัน โดยมีการวิเคราะห์โปรแกรมและตัดสินใจบางอย่างที่สามารถทำได้ในช่วงคอมไพล์ก่อนเพื่อให้โปรแกรมสามารถทำงานได้เร็วขึ้นก่อน และการใช้อินเทอร์พรีเตอร์กระทำการทำให้สามารถนำไปประมวลผลที่ใดก็ได้ที่มีอินเทอร์พรีเตอร์ติดตั้งอยู่ โดยจะพบในภาษาโปรแกรมส่วนมากและเป็นแบบที่ใช้โดยภาษา Java

ภาพที่ 1.3 แผนภาพแสดงการแปลภาษาโปรแกรมแบบผสม Compiler และ Interpreter เข้าด้วยกัน

คำบรรยายภาพ: ขั้นตอนการแปลภาษาโปรแกรมแบบผสม Compiler และ Interpreter

1. Source Program เป็นข้อมูลนำเข้าสำหรับประมวลผลโดย Translator
2. Translator วิเคราะห์และแปล Source Program ด้วยวิธีการ Compiler ได้ผลลัพธ์เป็น Intermediate Program
3. Intermediate Program และ Input เป็นข้อมูลนำเข้าของ Virtual machine
4. Virtual machine กระทำการ Intermediate Program เพื่อประมวลผล Input ด้วยวิธีการของอินเทอร์พรีเตอร์ให้ผลลัพธ์เป็น Output



ในการอธิบายแบบจำลองการแปลภาษารูปแบบผสมผสานนี้จะยกตัวอย่างจากภาษา Java โดยการเขียนโปรแกรมได้เป็น Source Program แล้ว โปรแกรมที่ได้จะถูกนำไปวิเคราะห์และแปลภาษาด้วย Translator ซึ่งทำหน้าที่เหมือน compiler โดยได้ผลลัพธ์เป็น Intermediate program โดยสำหรับภาษา Java จะเรียกว่า Byte code ส่วนของผลลัพธ์ที่ได้จะรับประกันได้ว่าโปรแกรมเขียนได้ถูกต้องตามหลักของภาษา แต่ว่าผลลัพธ์ที่ได้นี้ไม่ใช่ที่ได้อยู่ในรูปของภาษาเครื่องที่สามารถไปกระทำการได้เลย แต่ด้วยภาษา Java ถูกออกแบบตามแนวคิด write once, run anywhere กล่าวคือ เขียนครั้งเดียวสามารถนำไปกระทำการได้ทุกที่ (หมายถึง ระบบปฏิบัติการใดก็ได้) โดยจะต้องนำไปกระทำการในสภาพแวดล้อมพิเศษตามแนวคิดของ interpreter สำหรับภาษา Java คือ Virtual Machine ที่จะทำหน้าที่แปล byte code เพื่อกระทำการตามคำสั่งบนแพลตฟอร์มต่าง ๆ เพื่อประมวลผล Input ให้ได้ Output ต่อไป

โดยประโยชน์ในการเขียนโปรแกรมแบบนี้ คือ โปรแกรมที่ถูกเขียนขึ้นและแปลเป็น byte code จะเคลื่อนย้ายได้ง่าย สามารถจะนำไปกระทำการบนแพลตฟอร์มใดก็ได้ที่มี Virtual Machine ติดตั้งอยู่ โดยไม่ต้องมีการทำการคอมไพล์ใหม่ หรือมีงานเขียนโปรแกรมเพื่อปรับแต่งเพิ่มเติมอีก

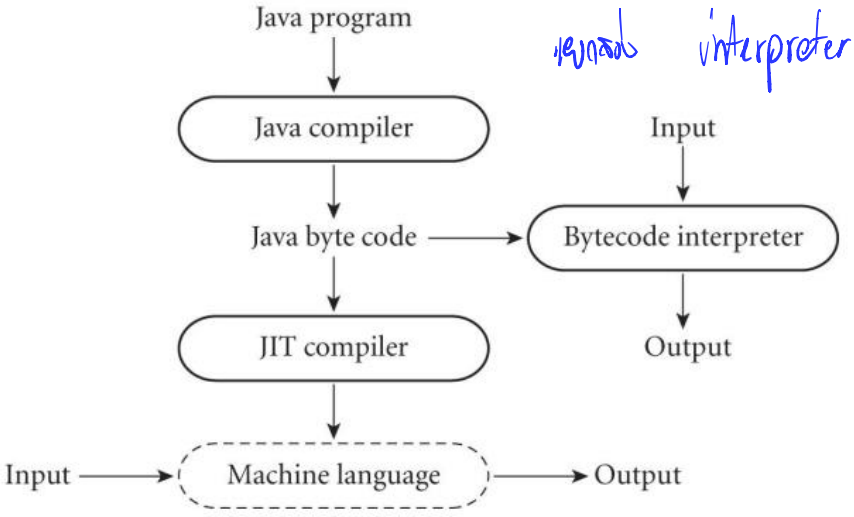
1.7 Just-in-Time Translation

เป็นการปรับปรุงเพิ่มสมรรถนะจากแบบผสมผสาน โดยเน้นที่ปัญหาของการที่ใช้อินเทอร์พรีเตอร์ในการกระทำการส่วนของ intermediate program จะมีส่วนที่ช้า คือ ต้องดำเนินการการแปลในทุกข้อความสั่ง ดังนั้นการแปล Just-in-Time จึงเกิดขึ้นเพื่อลดเวลาในส่วนของการแปลซ้ำ ๆ นี้

ภาพที่ 1.4 แผนภาพแสดงการขั้นตอนแปลภาษาโปรแกรมแบบ Just-in-Time

คำบรรยายภาพ: ขั้นตอนการแปลภาษาโปรแกรมแบบ Just-in-Time

1. Java Program เป็นข้อมูลนำเข้าสำหรับประมวลผลโดย Java Compiler
2. Java Compiler วิเคราะห์และแปล Java Program ได้ผลลัพธ์เป็น Java byte code
3. Java byte code เป็นข้อมูลนำเข้าสำหรับ JIT Compiler และ Bytecode interpreter
4. JIT Compiler วิเคราะห์เพื่อเลือกบางส่วนของ Java byte code เพื่อคอมไพล์เป็น Machine language
5. Input จะถูกนำเข้าเพื่อประมวลผล โดย Bytecode interpreter หรือ Machine language ขึ้นอยู่กับว่าส่วนของโค้ดที่จะประมวลผล Input นั้นถูกคอมไพล์ไว้หรือไม่
6. Bytecode interpreter ทำการแปลและกระทำการคำสั่งตาม Java byte code ให้ผลลัพธ์เป็น Output
7. Machine language กระทำการคำสั่ง ให้ผลลัพธ์เป็น Output (ไม่มีขั้นตอนการแปลภาษา)



ส่วนที่ปรับปรุงเพิ่มเติมขึ้นมา คือ ส่วนของ Just-in-Time (JIT) Compiler ที่จะมีหน้าที่ในการวิเคราะห์ Java byte code เพื่อหาว่ามีข้อความสั่งไหนถูกเรียกใช้บ่อย เนื่องจากในโปรแกรมจะมีข้อความสั่งที่ถูกกระทำการมากกว่า 1 ครั้ง อย่างเช่น ข้อความสั่งภายในลูป ส่วนคำสั่งเหล่าที่ถูกเรียกใช้บ่อยเหล่านี้จะถูกเลือกมาคอมไพล์เพื่อแปลเป็น Machine language เก็บไว้ในแคช (cache) และเมื่อการกระทำการมาถึงข้อความสั่งเหล่านี้ก็จะทำได้โดยทันทีโดยไม่ต้องทำขั้นตอนการแปลภาษา ในขณะที่ข้อความสั่งอื่นที่ไม่ถูกเรียกใช้บ่อยจะถูกกระทำการโดยอินเทอร์พรีเตอร์ที่ต้องมีขั้นการแปลภาษาก่อน ซึ่งจะทำให้การทำงานของโปรแกรมเร็วขึ้นตามจำนวนของงานแปลข้อความสั่งที่ลดลงไปได้ แต่จะเพิ่มเวลาเล็กน้อยในต้นเริ่มต้นของโปรแกรมในส่วนของการวิเคราะห์และแปลภาษาของ JIT compiler นั่นเอง

การทำงานแบบ JIT จะไม่ทำการคอมไพล์ทั้งโปรแกรม (Frequently used portion, rarely in full) แต่จะเลือกทำกับเฉพาะส่วนของโค้ดที่มีถูกเรียกใช้งานบ่อย ๆ ที่จะทำให้การทำงานของโปรแกรมรวดเร็วยิ่งขึ้นเมื่อเทียบกับการใช้อินเทอร์พรีเตอร์เพียงอย่างเดียว ยิ่งไปกว่านั้น หากทำการคอมไพล์ทั้งโปรแกรมการทำงานก็จะกลายเป็นแบบใช้ Compiler แทนซึ่งสำหรับภาษา Java แล้วจะกลายเป็นผิดวัตถุประสงค์ของภาษาที่ต้องการเขียนครั้งเดียวและใช้งานได้ทุกที่

B what is the diffent between machine code and byte code

Machine code and bytecode are both forms of low-level code used in computer systems, but they differ in their level of abstraction and their execution environments. Here's a breakdown of the differences between machine code and bytecode:

Machine Code:
Machine code is a low-level programming language that directly corresponds to the instructions executed by a computer's hardware. It consists of binary instructions specific to the processor architecture of the target machine. Each instruction represents a specific operation performed by the processor, such as arithmetic, data movement, or control flow. Machine code is written in a format that can be directly executed by the computer's CPU.

Key characteristics of machine code:

1. Platform-specific: Machine code is tied to a particular hardware architecture, such as x86, ARM, or MIPS. Programs written in machine code can only run on compatible processors.
2. Low-level: Machine code is written using binary instructions that directly interact with the hardware components. It provides a high level of control but requires a deep understanding of the underlying architecture.

Bytecode:
Bytecode is an intermediate representation of code that is executed by a virtual machine (VM). It is a form of compiled code that is not tied to a specific hardware architecture. Bytecode is usually generated from a higher-level programming language and can be executed on any platform that has a compatible virtual machine.

Key characteristics of bytecode:

1. Platform-independent: Bytecode is designed to be portable across different systems. It can be executed on any platform that has a virtual machine capable of interpreting the bytecode instructions.
2. Higher-level: Bytecode is generally closer to high-level programming languages than machine code. It may use more human-readable instructions and can provide additional features like automatic memory management, exception handling, and dynamic typing.

In summary, machine code is specific to a particular hardware architecture and directly executed by the computer's CPU. Bytecode, on the other hand, is an intermediate representation that is platform-independent and executed by a virtual machine.

B

so if i have program in cpp and i already complie it and send to another computer can they run it

If you have compiled a C++ program into machine code for a specific hardware architecture, you typically cannot directly run that compiled executable on a different computer with a different architecture. Machine code is specific to the underlying hardware, and different computer architectures have different instruction sets and binary formats.

To run a C++ program on a different computer, you generally have a few options:

1. Recompile on the target machine: You can recompile the C++ source code on the target machine using a C++ compiler compatible with that architecture. This ensures that the compiled program matches the target machine's instruction set and binary format.
2. Use a compatible virtual machine: If you have compiled the program into bytecode, which is platform-independent, you can run it on a compatible virtual machine that exists on the target machine. For example, if you have compiled the C++ program into Java bytecode, you can run it on a machine with a Java Virtual Machine (JVM) installed.
3. Cross-compilation: It is possible to cross-compile a program on one machine for a different target architecture. This involves using a cross-compiler that generates machine code for the target architecture. The resulting executable can then be transferred and run on the target machine.

It's worth noting that some platforms and programming frameworks provide additional tools or technologies for packaging and distributing applications in a way that simplifies deployment on multiple machines, such as containerization (e.g., Docker) or platform-specific app stores (e.g., Microsoft Store, Apple App Store).

In summary, if you have compiled a C++ program for a specific architecture, you typically need to recompile it on the target machine or use a compatible virtual machine to run it on a different computer.