JGROOVY: ส่วนต่อขยายตัวแปลภาษา JAVA เพื่อรองรับภาษา GROOVY
JGROOVY: EXTENSION OF JAVA COMPILER

ศิวดล เสถียรพัฒนากูล

SIWADOL SATEANPATTANAKUL

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาตรดุษฎีบัณฑิต
สาขาวิชาวิศวกรรมไฟฟ้า
คณะวิศวกรรมศาสตร์
สถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง
พ.ศ. 2555
KMITL-2012-EN-D-018-192

JGROOVY: GROOVY EXTENSION OF JAVA COMPILER

SIWADOL SATEANPATTANAKUL

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
DOCTOR OF ENGINEERING IN ELECTRICAL ENGINEERING
FACULTY OF ENGINEERING
KING MONGKUT'S INSTITUTE OF TECHNOLOGY LADKRABANG
2012

KMITL-2012-EN-D-018-192

หัวข้อวิทยานิพนธ์	JGROOVY: ส่วนต่อขยายตัวแปลภาษา JAVA เพื่อรองรับภาษา GROOVY
นักศึกษา	นายศิวดล เสถียรพัฒนากูล
รหัสประจำตัว	51060033
ระดับการศึกษา	วิศวกรรมศาสตรดุษฎีบัณฑิต
สาขาวิชา	วิศวกรรมไฟฟ้า
พ.ศ.	2555
อาจารย์ที่ปรึกษาวิทยานิพนธ์	ผศ.ดร. อรัญญา วลัยรัชต์

บทคัดย่อ

ภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไป (General Purpose Programming Languages, GPLs) ถูกออกแบบมาสำหรับแก้ปัญหางานในลักษณะภาพรวมซึ่งไม่สามารถตอบสนองกับปัญหาเฉพาะทางได้ นั่นเป็นเพราะว่าไม่ได้ถูกออกแบบมาเพื่อรองรับกับปัญหาใดปัญหาหนึ่ง ซึ่งการแก้ไขปัญหาที่มีลักษณะเฉพาะนั้นต้องใช้วิธีการและภาษาที่ถูกออกแบบและสร้างขึ้นมาโดยตรงที่เรียกว่าภาษาเฉพาะทาง (Domain Specific Languages, DSLs) ดังนั้นเพื่อตอบสนองต่อปัญหาเฉพาะทางที่หลากหลายและซับซ้อนทำให้เกิดการแก้ไขโครงสร้างของภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไป การแก้ไขโครงสร้างของภาษาในลักษณะนี้พบว่าทำให้ภาษานั้นๆ เกิดการเปลี่ยนแปลงซึ่งส่งผลให้ไม่สามารถตอบสนองต่อรากของภาษาเดิมได้อย่างถูกต้อง ซึ่งจะส่งผลให้เกิดผลกระทบต่อวิธีการและโครงสร้างข้อมูลที่ถูกพัฒนามาก่อนหน้า โดยพบว่าปัญหาที่เกิดขึ้นมานั้นเป็นเพราะโครงสร้างในตัววิเคราะห์คำ (Lexical Analyzer) ตัวตรวจสอบไวยากรณ์ (Syntactic Analyzer) และตัวตรวจสอบความหมาย (Semantic Analyzer) ไม่สามารถแยกความแตกต่างของโครงสร้างของภาษาที่ถูกขยายเพิ่มเติมได้เนื่องจากมีลักษณะคำสั่งที่ใกล้เคียงกัน

ดังนั้นงานวิจัยนี้ได้นำเสนอโครงสร้างของตัวแปลภาษาที่สามารถคัดแยกความแตกต่างของคำสั่งภาษาที่มีลักษณะใกล้เคียงกันด้วย Stack Machine Analyzer ในส่วนของตัววิเคราะห์คำ และสถาปัตยกรรมแบบขยายในส่วนของตัวตรวจสอบไวยากรณ์และตัวตรวจสอบความหมายเพื่อรองรับปัญหาที่เกิดขึ้นจากความใกล้เคียงกันของคำสั่งต่างๆ โดยใช้ภาษาที่มีความใกล้เคียงกันมาทำการขยายความสามารถโดยใช้ ภาษา Java และภาษา Groovy ซึ่งพบว่าภาษาและตัวแปลภาษา JGroovy ที่ถูกพัฒนามานั้นสามารถคัดแยกความแตกต่างของภาษา Java ที่เป็นภาษาหลักได้อย่างถูกต้อง อีกทั้งสามารถคัดแยกภาษา Groovy ที่ถูกขยายไว้อย่างถูกต้องด้วยเช่นกัน ซึ่งด้วยโครงสร้างของการขยายภาษาในลักษณะนี้สามารถเพิ่มทางเลือกสำหรับสร้างไบต์โค้ดได้อย่างเหมาะสมทำให้สามารถลดขนาดของไบต์โค้ดลงร้อยละ 8 ถึงร้อยละ12

Thesis Title	JGROOVY: GROOVY EXTENSION OF JAVA COMPILER
Student	Mr. Siwadol Sateanpattanakul
Student ID	51060033
Degree	Doctor of Engineering
Program	Electrical Engineering
Year	2555
Thesis Advisor	Asst. Prof. Dr. Aranya Walairacht

Abstract

General Purpose Programming languages (GPLs) are designed to solve common problems. They cannot explain the specific problem because GPLs did not be implemented to fix a specific problem. The way to solve the specific problem is the Domain Specific Languages (DSLs). For response many specific and complex problems, GPL has to change the language structure and rise to the new language. However, this approach makes directly side effect to the language. The new language cannot support the former language. The problem is the lexical analyzer, syntactic analyzer and semantic analyzer cannot separate the similar structure of the new language.

This research proposes the stack machine analyzer to support the similar syntax. And the extended approach to support the new language structure. This research proposes an extended architecture-technique to implement computer programming language and compiler through extending Java with Groovy Language. The extensible language is called "JGroovy". And JGroovy is supported both by Java and Groovy language. The compiler for JGroovy is JGroovy compiler (JGC), which produce more compatible byte code for Java source code than Javac can claim to be. Moreover, it also produces better compact byte code than Groovy compiler, with an approximate of 8 – 12 percent.

กิตติกรรมประกาศ

วิทยานิพนธ์เล่มนี้สำเร็จได้ด้วยความกรุณาจากอาจารย์ที่ปรึกษา ผศ.ดร.อรัญญา วลัยรัชต์ ที่ให้ความช่วยเหลือ ให้คำชี้แนะช่วยแก้ปัญหาตลอดจนให้ความรู้และประการณ์ที่ดีแก่ข้าพเจ้า

ขอขอบพระคุณ ผศ.ดร.สมศักดิ์ วลัยรัชต์ ที่ได้กรุณาให้คำแนะนำตลอดจนข้อชี้แนะ ที่คอยให้คำปรึกษาและชี้แนะแนวทางการทำวิจัยและทำการทดลองจนในที่สุดทำให้วิทยานิพนธ์ฉบับนี้สำเร็จลงได้

ขอขอบพระคุณ Prof. Dr. Kazuhiko Hamamoto ที่ได้กรุณาให้คำชี้แนะแนวการทดลองและเขียนบทความวิจัย รวมถึงห้องปฏิบัติการ ณ มหาวิทยาลัยโตไก ประเทศญี่ปุ่น

ทุนอุดหนุนโครงการพัฒนาอาจารย์ สาขาขาดแคลน หลักสูตรร่วมสถาบันกับมหาวิทยาลัยในต่างประเทศ (AUN/SEEN-NET) ที่ให้การสนับสนุนการวิจัยนี้ ขอขอบคุณพี่ๆ น้อง ๆ ในห้องปฏิบัติการทุกคนจากสถาบันเทคโนโลยีพระจอมเกล้าเจ้าคุณทหารลาดกระบัง และ มหาวิทยาลัยโตไก ประเทศญี่ปุ่น

สำหรับคุณงามความดีอันใดที่เกิดจากวิทยานิพนธ์ฉบับนี้ ข้าพเจ้าขอมอบให้กับบิดามารดา ซึ่งเป็นที่รักและเคารพยิ่ง ตลอดจนครูอาจารย์ที่เคารพทุกท่านที่ได้ประสิทธิ์ประสาทวิชาความรู้และถ่านทอดประสบการณ์ที่ดีให้แก่ข้าพเจ้า

ศิวดล เสถียรพัฒนากูล

สารบัญ

หน้า

หน้า

หน้า

สารบัญตาราง

ตารางที่ หน้า

ตารางที่ หน้า

ตารางที่ หน้า

ตารางที่ หน้า

ตารางที่ หน้า

สารบัญรูป

บทที่ 1
บทนำ

1.1 ความเป็นมา

ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์ เป็นภาษาที่ถูกออกแบบและทำการประดิษฐ์ขึ้นเพื่อแสดงให้เห็นถึงการทำงานที่สามารถถูกดำเนินการได้โดยเครื่องจักร โดยเฉพาะเครื่องคอมพิวเตอร์ ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์สามารถถูกใช้สำหรับสร้างโปรแกรม หรือระบบการทำงานได้ ซึ่งเป็นลักษณะของการควบคุมพฤติกรรมการทำงานของเครื่องจักร ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์หลายภาษามีลักษณะรูปแบบภาษาเป็นลักษณะเฉพาะตัวซึ่งถูกกำหนดลักษณะและการทำงานด้วยไวยากรณ์ (Syntax) และ ความหมาย (Semantics) แต่ภาษาบางชนิดนั้นบางครั้งอาจจะสามารถถูกระบุได้ด้วยข้อจำกัดที่อยู่ในรูปแบบของเอกสาร อย่างเช่น ภาษาซีเป็นภาษาที่ถูกระบุความสามารถเอาไว้โดยมาตรฐานของ ISO แต่ภาษาอย่างเช่น Perl เป็นภาษาที่มีอิสระในการถูกพัฒนามากกว่าเพราะไม่ได้ใช้มาตรฐานในระดับสากลเท่าภาษาซี

โดยเริ่มแรกก่อนการประดิษฐ์ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์นั้น การติดต่อสื่อสารกับเครื่องจักรต้องถูกกระทำโดยตรงจากมนุษย์ หรือแผ่นคำสั่งเพื่อสั่งการทำงาน อย่างเช่น งานประเภทของเครื่องจักรทอผ้า และเครื่องเล่นเปียโน แต่หลังจากนั้นมาเป็นต้นมา ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์ถูกสร้างขึ้นมาอย่างมากมาย โดยเฉพาะเพื่อตอบสนองในงานทางด้านของคอมพิวเตอร์ โดยลักษณะของภาษาสำหรับเขียนโปรแกรมคอมพิวเตอร์ส่วนมากอธิบายการคำนวณในลักษณะเป็นลำดับของคำสั่งตามความลักษณะของการทำงาน (Sequence of commands) แต่ภาษาบางอย่างที่สนับสนุนการทำงานแบบ Functional programming หรือ Logic programming จะใช้ลักษณะการทำงานที่แตกต่างออกไป โดยภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์นั้นเปรียบได้กับเป็นสัญลักษณ์ของภาษาที่ไว้สำหรับติดต่อกับเครื่องคอมพิวเตอร์โดยกำหนดด้วยวิธีของการคำนวณหรือขั้นตอนวิธี แต่ในบางความหมายได้จำกัดความของคำว่า “ภาษาสำหรับการเขียนโปรแกรม” ว่าเป็นภาษาที่สามารถแสดงขั้นตอนวิธีที่เป็นไปได้ทั้งหมดของอัลกอริธึม [1]

ลักษณะของภาษาสำหรับการพัฒนาโปรแกรมนั้นสามารถถูกแบ่งออกมาเป็น 2 ลักษณะได้แก่ ภาษาสำหรับการพัฒนาโปรแกรมเฉพาะทาง (Domain Specific Languages, DSLs) ซึ่งใช้เพื่อแก้ไขลักษณะปัญหาที่เกิดจากลักษณะงานเฉพาะด้าน [2] อย่างเช่น ภาษา SQL นั้นถูกสร้างมาเพื่อติดต่อและทำงานทางด้านฐานข้อมูล หรือภาษา Mata ซึ่งเป็นภาษาที่ถูกสร้างมาเพื่อพัฒนาโปรแกรมทางด้านเมตริกเป็นต้น และภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไป (General Purpose Programming Languages, GPLs) จนถึงปัจจุบันนี้มีภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปถูกสร้างขึ้นมาอย่างมากมายเพื่อตอบสนองวิธีการพัฒนาโปรแกรมตามความถนัด ข้อจำกัดต่างๆ รวมไปถึงสถาปัตยกรรมของเครื่องคอมพิวเตอร์ ซึ่งแน่นอนว่าภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปนั้นไม่ได้ถูกสร้างรวมกับภาษาที่ใช้ในการทำงานเฉพาะทางด้วยเช่นกัน ยกตัวอย่างเช่น ภาษาซี (C Programming Language) นั้นถูกออกแบบมาเพื่อถ่ายทอดคำสั่งการทำงานเพื่อติดต่อเครื่อง

คอมพิวเตอร์อย่างมีประสิทธิภาพโดยจะมีการเปลี่ยนคำสั่งเหล่านั้นเป็นภาษาเครื่อง (Assembly Language) ที่เหมาะสมสำหรับแต่ละสถาปัตยกรรม เป็นต้น ดังนั้นเพื่อตอบสนองการทำงานที่เกิดขึ้นเมื่อมีการติดต่อกับลักษณะงานเฉพาะทาง จึงถูกออกแบบมาเพื่อรวมลักษณะการทำงานของลักษณะการทำงานของ 2 วิธีเข้าด้วยกัน โดยการเขียนโปรแกรมเชิงภาษา (Language Oriented Programming, LOP) จะประกอบไปด้วยกลุ่มของภาษาสำหรับการพัฒนาโปรแกรมเฉพาะทาง ซึ่งจะทำงานในภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปหนึ่งชนิด [3]

ภาษา Java เป็นหนึ่งในภาษาสำหรับพัฒนาโปรแกรมแบบทั่วไป ซึ่งภาษา Java นั้นเป็นภาษาสำหรับการพัฒนาโปรแกรมเชิงวัตถุ (Object-Oriented Programming, OOP) ที่ทำงานบน Java Virtual Machine (JVM) ซึ่งเป็นแพลตฟอร์มที่สนับสนุนชุดคำสั่งสำหรับทำงานบนสถาปัตยกรรมต่างๆ ซึ่งทำให้ JVM เองเป็นหนึ่งส่วนประกอบหลักที่สำคัญสำหรับการพัฒนาโปรแกรมคอมพิวเตอร์ [4] ด้วยลักษณะดังที่กล่าวมาข้างต้นทำให้ภาษาสำหรับพัฒนาโปรแกรมแบบทั่วไปแบบอื่นๆ ต้องการที่จะมาทำงานบนแพลตฟอร์มของ JVM ด้วยเช่นกัน ยกตัวอย่างเช่น ภาษา Ada ซึ่งตัวแปลภาษาของภาษา Ada นั้นสามารถแปลโค้ดต้นฉบับ (Source code) ให้สามารถทำงานบนแพลตฟอร์มของ [5] JVM ได้ เป็นต้น อีกทั้งภาษา Java เองถูกออกแบบมาให้สามารถแก้ไข ดัดแปลง และเพิ่มเติมตัวลักษณะภาษาได้ ซึ่งทางภาษา Java นั้นได้มีมาตรฐานสำหรับการออกแบบภาษา Java เพื่อเป็นบรรทัดฐานสำหรับการพัฒนาภาษาสำหรับการเขียนโปรแกรม ด้วยเหตุนี้เองทำให้มีภาษาสำหรับการพัฒนาโปรแกรมนำภาษา Java มาทำการขยายความสามารถเพื่อรองรับการพัฒนาโปรแกรมในรูปแบบต่างๆ อย่างหลากหลาย ซึ่งการขยายความสามารถของภาษา Java นั้นสามารถทำได้ใน 2 ลักษณะคือการขยายแบบภายใน (Internal extended) ซึ่งเป็นการแก้ไขภาษาแบบทางอ้อม และการขยายแบบภายนอก (External extended) ซึ่งเป็นการแก้ไขลักษณะของภาษาโดยตรง การขยายความสามารถของภาษา Java โดยทางอ้อมนั้นจะไม่ส่งผลกระทบต่อวิธีการพัฒนาโปรแกรมภาษา Java เพราะว่าเป็นการสร้างชุดคำสั่งขึ้นมาเพื่อสนับสนุนการทำงานในลักษณะนั้นๆ ภายใต้รูปแบบของภาษาเดิม ยกตัวอย่างเช่นการติดต่อกับฐานข้อมูล ซึ่งจะมาการสร้างชุดคำสั่งเพื่อมาจัดการการเข้าถึงฐานข้อมูลภายใต้ลักษณะวิธีการพัฒนาโปรแกรมของภาษา Java หรือการเพิ่มความสามารถของภาษาอื่นๆเพิ่มเติมเข้ามาอย่างเช่น JRuby ซึ่งจะสร้างชุดคำสั่งเพื่อตอบสนองการทำงานที่เพิ่มเติมเข้ามาเช่นกัน [6] ซึ่งการขยายความสามารถของภาษาในลักษณะนี้บ่อยครั้งทำให้มีข้อจำกัด และเพิ่มความซับซ้อนในการพัฒนาโปรแกรมมากขึ้น ดังนั้นเพื่อตอบสนองต่อความต้องการทางด้านการจัดการงานทางด้านเฉพาะทางอย่างตรงไปตรงมาต้องทำการขยายความสามารถของภาษาโดยตรง ยกตัวอย่างเช่น ภาษา JR ซึ่งภาษา JR นี้ถูกออกขยายออกมาจากภาษา Java เพื่อสนับสนุนและรองรับการทำงานทางด้านการทำงานในภาวะพร้อมกัน (Concurrent programming language) [7] หรือภาษา JMatch ซึ่งเป็นภาษาที่ถูกออกแบบมาเพื่อสนับสนุนการทำงานทางด้านลักษณะชนิดของข้อมูลเพื่อให้เกิดการเข้าถึงข้อมูลอย่างสอดคล้องกัน [8] เป็นต้น โดยการแก้ไขในลักษณะนี้จะส่งผลให้เกิดการเปลี่ยนแปลงของลักษณะภาษาโดยตรง โดยอาจมีการเพิ่มคำเฉพาะ (Reserve word) ขึ้นมาส่งผลให้วิธีนี้สามารถตอบสนองต่อการแก้ไขปัญหาได้โดยตรงและมีประสิทธิภาพ อย่างไรก็ตามการขยายการทำงานในลักษณะนี้ต้องใช้ทรัพยากรในการแก้ไขอย่างมากโดยเฉพาะทางด้านเวลา ดังนั้นภาษา Groovy จึงถูกพัฒนาขึ้นมาโดยจัดการงานที่ติดต่อกับงานเฉพาะทางโดยเฉพาะ โดยทางภาษา Groovy จะมีฟังค์ชั่นพิเศษที่เรียกว่าช่องทางปิด (Closure) เพื่อใช้ติดต่อกับงานที่ต้องการอ้างอิงถึง [9] ซึ่งจะสามารถติดต่อได้โดยใช้วิธีการแก้ไขปัญหาเฉพาะทางได้โดยตรง โดยไม่ขึ้นอยู่กับลักษณะของ

ภาษา Java การขยายความสามารถในลักษณะนี้ต้องใช้วิธีการขยายความสามารถของภาษา Java โดยตรงซึ่งการขยายความสามารถในลักษณะนี้จะส่งผลกระทบต่อภาษา Java อย่างหลีกเลี่ยงไม่ได้ ภาษา Groovy มีลักษณะการเขียนโปรแกรมที่ใกล้เคียงกับภาษา Java แต่เพื่อตอบสนองต่อการเขียนโปรแกรมที่กระชับ และฟังค์ชั่นพิเศษนี้เองส่งผลทำให้ภาษา Groovy ไม่สามารถสนับสนุนต่อวิธีการพัฒนาโปรแกรมของภาษา Java ได้อย่างถูกต้องทั้งหมด

1.2 ความสำคัญของปัญหา

การขยายความสามารถของภาษาเป็นหนึ่งในวิธีการตอบสนองต่อการแก้ไขลักษณะปัญหาซึ่งเกิดขึ้นจากการติดต่อกับลักษณะงานที่มีลักษณะเฉพาะตัว เพราะไม่อาจจะแก้ไขปัญหาเหล่านั้นด้วยวิธีการหรือช่องทางแบบปรกติ ดังนั้นเพื่อตอบสนองต่อลักษณะงานเฉพาะทางต่างๆ ทำให้ไม่อาจที่จะเลี่ยงต่อการขยายความสามารถของภาษาไปได้ การขยายความสามารถของภาษาโดยตรงเพื่อตอบสนองต่อการแก้ไขปัญหาที่หลากหลายนั้นส่งผลกระทบโดยตรงต่อลักษณะการทำงานของภาษา ซึ่งทำให้ลักษณะและวิธีการพัฒนาโปรแกรมเกิดการเปลี่ยนแปลงทั้งทางตรงและทางอ้อม โดยผลกระทบทางตรงนั้นอาจทำให้โค้ดต้นฉบับที่ถูกเขียนพัฒนามาก่อนหน้าไม่สามารถตอบสนองการทำงานได้อย่างถูกต้องส่งผลให้ต้องมีการแก้ไขโค้ดต้นฉบับ หรือพัฒนาโค้ดต้นฉบับขึ้นมาใหม่

เพื่อสนับสนุนการทำงานของภาษา Java อย่างมีประสิทธิภาพ และป้องกันซึ่งรวมไปถึงลดผลการทบต่อการตอบสนองของการขยายความสามารถของภาษาจึงเป็นสิ่งที่จำเป็นอย่างหลีกเลี่ยงไม่ได้

1.3 วัตถุประสงค์

1. นำเสนอการขยายความสามารถของภาษาใหม่ โดยใช้ภาษา Java เป็นภาษาหลัก
2. นำเสนอวิธีการแก้ไขปัญหาที่เกิดจากลักษณะของภาษาที่มีความใกล้เคียงกัน
3. นำเสนอวิธีการเพื่อรองรับจากการขยายความสามารถของภาษาในส่วนของไวยากรณ์
4. นำเสนอวิธีการเพื่อรองรับจากการขยายความสามารถของภาษาในส่วนของความหมาย
5. เพื่อศึกษาผลกระทบต่อการขยายความสามารถของภาษาใหม่ต่อโครงสร้างภาษาเดิม

1.4 ผลงาน

เพื่อลดผลกระทบจากการตอบสนองต่อการขยายความสามารถของภาษา Java งานวิจัยนี้นำเสนอวิธีการออกแบบภาษา และพัฒนาตัวแปลภาษาที่สามารถตอบสนองต่อการเพิ่มลักษณะภาษาที่มีความใกล้เคียงกัน เพื่อลดผลกระทบต่อการใกล้เคียงกันของภาษาที่ไม่สามารถแบ่งแยกในระดับของไวยากรณ์ ทำให้ต้องมีการคัดแยกลักษณะของภาษาตั้งแต่ในส่วนของการวิเคราะห์คำ (Lexical Analysis) โดยในการวิจัยนี้ได้นำเสนอภาษา Groovy ที่สามารถตอบสนองและจัดการงานทางด้านเฉพาะทางได้อย่างหลากหลายซึ่งมีลักษณะของภาษาคล้ายคลึงกับภาษา Java เพื่อเป็นชี้วัดต่อการขยายความสามารถของภาษา Java ในลักษณะนี้ รวมถึงเพื่อเพิ่มและสนับสนุนความสามารถของภาษา Java ต่อการจัดการงานทางเฉพาะทางให้เหมาะสมยิ่งขึ้น

1.5 ขอบเขตของงานวิจัย

ขอบเขตของงานวิจัยนี้จะเสนอวิธีการในการออกแบบภาษาโปรแกรมใหม่กับภาษา Java ซึ่งเป็นภาษาสำหรับการเขียนโปรแกรมที่ถูกใช้กันอย่างแพร่หลายได้โดยใช้วิธีการคัดแยกลักษณะของภาษาที่คล้ายคลึงกันมากตั้งแต่ในส่วนของการวิเคราะห์คำ (Lexical Analysis) เพื่อแบ่งเบาภาระในขั้นตอนของไวยากรณ์ของภาษา และศึกษาผลกระทบต่อการขยายความสามารถของภาษา และการทำงานของตัวแปลภาษาต่อการขยายความสามารถของภาษา รวมไปถึงลักษณะของไบต์โค้ดเมื่อถูกเรียกใช้งานในลักษณะงานเดิม และลักษณะงานใหม่ที่ถูกเพิ่มเติมความสามารถเข้ามา

1.6 งานวิจัยในส่วนต่างๆ

งานวิจัยในฉบับนี้ประกอบไปด้วยส่วนต่างๆ ดังต่อไปนี้ บทที่ 2 เป็นการอธิบายถึงงานวิจัย และทฤษฎีต่างๆ ที่เกี่ยวข้องกับงานวิจัยซึ่งจะประกอบไปด้วยส่วนของการวิเคราะห์คำ (Lexical Analysis) ส่วนของการตรวจสอบไวยากรณ์ (Syntactic Analysis) และส่วนตรวจสอบความหมายของภาษา (Semantic Analysis) [10] ในส่วนของบทที่ 3 เป็นการอธิบายถึงการแก้ไขปัญหาที่เกิดจากการขยายความสามารถของภาษาที่มีลักษณะคล้ายคลึงกัน จากนั้นในส่วนของบทที่ 4 จะเป็นการทดสอบผลกระทบต่อการขยายความสามารถของภาษาต่อโครงสร้างภาษา Java เดิม และภาษาใหม่ที่ถูกขยายเพิ่มเข้ามา บทที่ 5 ซึ่งเป็นบทสุดท้ายจะเป็นการสรุปผล และแนวโน้มของงานในอนาคต

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

ในส่วนของบทที่ 2 นี้จะประกอบไปด้วยการอธิบายถึงทฤษฎีที่ใช้ในการทำงานด้านการออกแบบภาษาและตัวแปลภาษา ซึ่งแน่นอนว่างานทางด้านการออกแบบภาษานั้นจะต้องคำนึงถึงหน้าที่การทำงานที่เหมาะสมของภาษาแต่ละลักษณะ รวมถึงการสร้างตัวแปลภาษาสำหรับภาษาที่ใช้ในการเขียนโปรแกรมนั้นต้องถูกจัดการด้วยโครงสร้างที่จำเพาะในแต่ละส่วน ซึ่งจะมีวิธีการที่แตกต่างกันโดยแต่ละวิธีจะขึ้นอยู่กับข้อบังคับของโครงสร้างภาษาที่ได้ถูกออกแบบไว้ในข้างต้น

การออกแบบโครงสร้างภาษาให้เหมาะสมต่อการนำมาขยายความสามารถเพิ่มเติมลงไปนั้น ส่วนของโครงสร้างของภาษาจำเป็นต้องมีความยืดหยุ่นหรือง่ายต่อการนำมาขยาย และเหมาะสมต่อการนำไปใช้ให้มากที่สุด ซึ่งจำเป็นต้องมีการค้นคว้า อ้างอิง และอธิบายถึงงานวิจัยที่เกี่ยวข้องในส่วนต่างๆ จึงได้มีการศึกษาเพื่อนำไปใช้ในการออกแบบภาษา และพัฒนาตัวแปลภาษาที่เหมาะสม

2.1 ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์

ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์เป็นภาษาที่ถูกประดิษฐ์เพื่อตอบสนองทางด้านการส่งคำสั่งต่างๆ สำหรับใช้ในการควบคุมการทำงานของเครื่องจักรโดยเฉพาะเครื่องคอมพิวเตอร์ ซึ่งแน่นอนว่าภาษาสำหรับการพัฒนาโปรแกรมคอมพิวเตอร์นี้สามารถนำมาพัฒนาโปรแกรมและแอปพลิเคชั่นเพื่อควบคุมและตอบสนองต่อรูปแบบการงานของเครื่องคอมพิวเตอร์ได้อย่างหลากหลายตามแต่ลักษณะของอัลกอริทึม [1, 10]

โดยเริ่มแรกก่อนการประดิษฐ์ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์นั้น การติดต่อสื่อสารกับเครื่องจักรต้องถูกกระทำโดยตรงจากมนุษย์ หรือแผ่นคำสั่งเพื่อกำหนดคำสั่งสำหรับการทำงาน อย่างเช่น งานประเภทของเครื่องจักรทอผ้า และเครื่องเล่นเปียโน แต่หลังจากนั้นมาเป็นต้นมาภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์ถูกสร้างและพัฒนาขึ้นมาอย่างมากมาย โดยเฉพาะเพื่อตอบสนองในงานทางด้านของคอมพิวเตอร์ โดยลักษณะของภาษาสำหรับเขียนโปรแกรมคอมพิวเตอร์ส่วนมากอธิบายการคำนวณในลักษณะเป็นลำดับของคำสั่งตามความลักษณะของการทำงาน

ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์นั้นเปรียบได้กับสัญลักษณ์ที่ไว้สำหรับใช้ในการติดต่อระหว่างมนุษย์และเครื่องคอมพิวเตอร์โดยจะถูกกำหนดด้วยวิธีของการคำนวณหรือขั้นตอนวิธี แต่ในบางความหมายได้จำกัดความของคำว่า “ภาษาสำหรับการเขียนโปรแกรม”. ว่าเป็นภาษาที่สามารถแสดงขั้นตอนวิธีที่เป็นไปได้ทั้งหมดของอัลกอริธึม โดยลักษณะการพิจาณาที่สำคัญที่ไว้ใช้สำหรับการพิจารณาภาษาของการเขียนโปรแกรมได้แก่

- หน้าที่การทำงานและเป้าหมาย โดยภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์เป็นภาษาที่ใช้สำหรับเขียนโปรแกรมคอมพิวเตอร์โดยเกี่ยวข้องกับการคำนวณบางประเภทหรืออัลกอริธึมและอาจจะมีการควบคุมอุปกรณ์ภายนอกอย่างเช่น เครื่องพิมพ์ เครื่องจานแม่เหล็ก หุ่นยนต์ หรืออื่นๆ ยกตัวอย่างภาษา PostScript ซึ่งโดยส่วนมากจะถูกใช้

สำหรับสร้างโปรแกรมอื่นๆเพื่อควบคุมการทำงานของเครื่องพิมพ์ หรือการแสดงค่า นอกเหนือไปจากนี้ภาษาสำหรับการเขียนโปรแกรมยังถูกใช้สำหรับบางอย่างที่เป็นนามธรรม เป็นที่ยอมรับกันแล้วว่าคุณสมบัติของภาษาสำหรับการเขียนโปรแกรมที่ครบถ้วนนั้นต้องมีคำอธิบายซึ่งประกอบได้ด้วยเครื่องมือหรือโปรเซสเซอร์สำหรับภาษาในรูปของบริบทจริงของภาษาสำหรับการเขียนโปรแกรมที่เกี่ยวข้องกับคอมพิวเตอร์รับรู้ได้ ว่าเป็นภาษาที่แตกต่างจากภาษาปกติ โดยทั่วๆ ไปภาษาปกตินั้นถูกใช้เพื่อสื่อสารกับบุคคลทั่วๆไป แต่ในขณะที่ภาษาสำหรับการเขียนโปรแกรมนั้นถูกใช้สำหรับติดต่อสื่อสารกับเครื่องจักร

- รูปแบบนามธรรม โดยภาษาสำหรับเขียนโปรแกรมนั้นจะมีส่วนนามธรรมไว้สำหรับการกำหนดและจัดการโครงสร้างของข้อมูล หรือควบคุมการทำงานของระบบ โดยส่วนที่สำคัญคือภาษาสำหรับการเขียนโปรแกรมนั้นต้องสนับสนุนส่วนนามธรรมตามทฤษฎีอย่างเพียงพอ โดยตามทฤษฎีนั้นเป็นตัวที่กำหนดสำหรับโปรแกรมเมอร์เพื่อทำการจัดการส่วนนามธรรมอย่างเหมาะสม
- ความสามารถในการจัดการคำสั่ง ซึ่งตามทฤษฎีของการคำนวณนั้นจะสามารถแยกภาษาโดยการคำนวณความสามารถในการจัดการกับสำนวนโดยภาษาสำหรับการเขียนโปรแกรมที่สมบูรณ์ต้องสามารถทำงานอัลกอริธึมได้ แต่สำหรับภาษาที่ไม่สามารถทำได้ อย่างเช่น ภาษาANSI/ISO ภาษาSQL นั้นจะไม่ถูกเรียกว่าเป็นภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์

ภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์แต่ละชนิดจะมีลักษณะรูปแบบภาษาเป็นลักษณะเฉพาะตัวที่แตกต่างกันออกไป โดยจะถูกกำหนดลักษณะและคำจำกัดความของภาษาด้วยไวยากรณ์และความหมาย แต่ภาษาบางชนิดนั้นบางครั้งอาจจะถูกระบุได้ด้วยข้อจำกัดที่อยู่ในรูปแบบของเอกสารที่เป็นมาตรฐาน อย่างเช่น ภาษาซี (C Programing Language) เป็นภาษาที่ถูกระบุคุณลักษณะเอาไว้โดยมาตรฐานขององค์การมาตรฐานสากล หรือองค์การระหว่างประเทศว่าด้วยการมาตรฐาน (International Standardization and Organization, ISO) แต่ภาษาบางชนิดเช่น Perl จะเป็นภาษาที่มีอิสระในการถูกพัฒนามากกว่าเพราะไม่ได้ถูกกำหนดมาตรฐานของภาษาให้อยู่ในระดับสากลเท่าภาษาซี เป็นต้น

อย่างที่ได้กล่าวไว้ข้างต้นว่าภาษาสำหรับการเขียนโปรแกรมนั้นตอบสนองการทำงานในลักษณะที่ต่างกันออกไป ซึ่งลักษณะการตอบสนองในลักษณะที่ต่างกันออกไปนี้ทำให้เกิดวิธีการเขียนโปรแกรมออกมาในรูปแบบ และวิธีที่ต่างกันโดยปกติวิธีการเขียนโปรแกรมโดยทั่วๆ ไปนั้นมีอยู่ 6 ชนิดได้แก่ วิธีการเขียนโปรแกรมแบบ Imperative วิธีการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming) วิธีการเขียนโปรแกรมเชิงคู่ขนาน (Concurrent Programming) วิธีการเขียนโปรแกรมเชิงการทำงาน (Functional Programming) วิธีการเขียนโปรแกรมเชิงตรรกะ (Logic Programming) และ วิธีการเขียนโปรแกรมแบบสคริปต์ (Script Programming) [1]

ภาษาสำหรับการเขียนโปรแกรมทุกภาษาจะมีไวยากรณ์ ความหมาย และรูปแบบการเขียนของตัวเองจะที่เห็นว่าภาษาที่มนุษย์ใช้นั้นจะมีไวยากรณ์ ความหมายที่แตกต่าง แต่รูปแบบการเขียนนั้นจะมีเฉพาะภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์

- ไวยากรณ์ของภาษาสำหรับเขียนโปรแกรมคอมพิวเตอร์นั้นเกี่ยวข้องกับรูปแบบของการเขียนโปรแกรม เนื่องจากว่าวิธีการจัดการคำสั่ง การประกาศตัวแปร และการจัดการงานในรูปแบบอื่นๆ นั้นจะต้องถูกเตรียมให้อยู่ในรูปแบบที่เข้าใจง่าย
- ความหมายของภาษาสำหรับเขียนโปรแกรมคอมพิวเตอร์นั้นเกี่ยวข้องกับความหมายของการจัดการโปรแกรม โดยจะต้องสามารถตีความหมายได้ง่ายเมื่อต้องทำงานกับคอมพิวเตอร์
- รูปแบบการเขียนของภาษาสำหรับเขียนโปรแกรมคอมพิวเตอร์นั้นจะต้องแสดงให้เห็นถึงมุ่งหมายที่จะใช้ในการปฏิบัติและใช้งาน

ซึ่งเบื้องต้นนั้นพบว่าไวยากรณ์ของภาษานั้นจะมีอิทธิพลต่อรูปแบบการเขียนอ่านโปรแกรมจากโปรแกรมเมอร์อีกคน และอ่านโดยโปรแกรมเมอร์คนอื่นๆ และคัดแยกโดยเครื่องคอมพิวเตอร์ ในส่วนของความหมายนั้นจะบ่งบอกถึงวิธีการที่โปรแกรมถึงดำเนินการโปรแกรมเมอร์อีกคน และสามารถทำความเข้าใจด้วยโปรแกรมเมอร์อีกคนซึ่งจะถูกตีความโดยเครื่องคอมพิวเตอร์ สุดท้ายคือรูปแบบการเขียนซึ่งมีอิทธิพลต่อวิธีการเขียนโปรแกรมซึ่งจะเกี่ยวเนื่องกับการออกแบบและพัฒนาโปรแกรม

โดยทั่วๆ ไปภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์สามารถถูกแบ่งตามลักษณะการแก้ปัญหาลักษณะงานได้เป็น 2 ชนิดคือ ภาษาสำหรับการพัฒนาโปรแกรมเฉพาะทาง (Domain Specific Languages, DSLs) และ ภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไป (General Purpose Programming Languages, GPLs)

2.1.1 ภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไป (GPLs)

ภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปนั้นใช้สำหรับพัฒนาซอฟต์แวร์ในลักษณะงานที่หลากหลาย โดยทั่วไปจะเป็นการแก้ปัญหาที่มีลักษณะคล้ายกัน ซึ่งข้อจำกัดของภาษาในลักษณะนี้คือจะไม่สามารถตอบสนองต่อปัญหาที่มีลักษณะเฉพาะเจาะจงมากๆ ได้ ยกตัวอย่างเช่นงานทางด้านฐานข้อมูล ซึ่งต้องใช้วิธีการและภาษา Structure Query Language (SQL) มาทำการจัดการปัญหาลักษณะดังกล่าว แน่นอนว่าภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปนั้นถูกคิดค้นและประดิษฐ์ขึ้นมาใช้อย่างมากมายเพื่อตอบสนองต่อลักษณะวิธีการและสถาปัตยกรรมที่แตกต่างกัน อย่างเช่น ภาษาซี (C Programming Language) นั้นถูกออกแบบมาเพื่อถ่ายทอดคำสั่งการทำงานสำหรับเพื่อติดต่อกับเครื่องคอมพิวเตอร์อย่างมีประสิทธิภาพ โดยจะมีการเปลี่ยนคำสั่งเหล่านั้นเป็นภาษาเครื่อง (Assembly Language) ที่เหมาะสมสำหรับแต่ละสถาปัตยกรรม เป็นต้น ซึ่งแน่นอนว่าการทำงานของภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปในลักษณะนี้อย่างเดียวไม่สามารถแก้ปัญหาที่เกิดจากงานเฉพาะทางได้ทั้งหมด

2.1.2 ภาษาสำหรับการพัฒนาโปรแกรมแบบเฉพาะทาง (DSLs)

ภาษาสำหรับการพัฒนาโปรแกรมแบบเฉพาะทางนี้ถูกออกแบบมาเพื่องานในลักษณะใดลักษณะหนึ่งสำหรับจัดการปัญหาด้วยรูปแบบและลักษณะของภาษาที่มีความเฉพาะเจาะจง ซึ่งแนวคิดของภาษานี้ไม่ได้เป็นการสร้างภาษาสำหรับการพัฒนาโปรแกรมแบบใหม่ โดยข้อจำกัดของภาษาแต่ละชนิดนั้นจะคงอยู่ไม่ได้ถูกแก้ไขแต่อย่างใด

ตัวอย่างของภาษาสำหรับการพัฒนาโปรแกรมแบบเฉพาะทางที่แสดงให้อย่างแพร่หลายภาษาหนึ่งคือ HyperText Markup Language (HTML) ที่ใช้สำหรับแสดงงานทางด้านเว็บเพจ ภาษา SQL สำหรับงานทางด้านฐานข้อมูล Yet Another Compiler Compiler (YACC) สำหรับงานทางด้านตัวแปลภาษาซึ่งจะทำงานคู่กับ Lex [11] หรือภาษาที่สำหรับการจัดการงานทางด้านแผงภาพอย่าง Generic Eclipse Modeling System [12] นั้นก็ถูกเรียกว่าเป็นภาษาสำหรับการพัฒนาโปรแกรมแบบเฉพาะทางด้วยเช่นกัน เป็นต้น

ภาษาสำหรับการพัฒนาโปรแกรมแบบเฉพาะทางถูกสร้างมาเพื่อแก้ปัญหาในโดเมนเฉพาะและไม่ได้มีวัตถุประสงค์อื่นที่อยู่นอกเหนือจากโดเมนของมันเอง ดังนั้นภาษาในลักษณะนี้จะไม่สนับสนุนการแก้ปัญหาที่อยู่ในโดเมนอื่น ซึ่งภาษาในลักษณะนี้บางครั้งเองอาจถูกเรียกว่าภาษาสำหรับการเขียนโปรแกรมขนาดเล็กหรือภาษาสคริปต์ (Script Language) ซึ่งภาษาเหล่านี้เองมีความใกล้เคียงกับภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปด้วยเช่นกัน

เพื่อตอบสนองต่อการแก้ปัญหาเฉพาะทางการเขียนโปรแกรมเชิงภาษา (LOP) จึงถูกนำเสนอเพื่อประยุกต์รูปแบบของลักษณะของภาษาทั้ง 2 ชนิดเข้าด้วยกัน โดยการเขียนโปรแกรมเชิงภาษานี้จะประกอบไปด้วยสำหรับการพัฒนาโปรแกรมแบบทั่วไปหนึ่งตัว และอาจจะมีกลุ่มของภาษาสำหรับการพัฒนาโปรแกรมแบบเฉพาะทางซึ่งอาจจะมีมากกว่า 1 ภาษา [3]

2.2 การออกแบบภาษา (Language design)

แนวคิดของลักษณะข้อมูล ชนิดของข้อมูล ตัวแปร การจัดเก็บ การผูกค่า และขอบเขตซึ่งรวมถึงขั้นตอนนามธรรมนั้นจะถูกพบอยู่ในภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์เกือบทุกภาษา แนวคิดนี้ทำให้ภาษาสำหรับการเขียนโปรแกรมมีการใช้คุณสมบัติร่วมกันกับภาษาปกติที่เกี่ยวข้องกับเพื่อวัตถุประสงค์สำหรับช่องทางในการติดต่อสื่อสาร โดยมีรูปแบบโครงสร้างของไวยากรณ์ ที่แยกออกจากส่วนของความหมายนั้นแสดงให้เห็นความเกี่ยวพันกันของภาษาหนึ่งๆ กับภาษาอื่น อย่างไรก็ตามสำหรับวิธีการเขียนโปรแกรมเชิงการทำงานนั้นแสดงให้เห็นถึงประโยชน์ของภาษาที่สามารถแก้ไขด้วยการกำหนดลักษณะการทำงาน โดยไม่ต้องเกี่ยวข้องกับการจัดการทางด้านตัวแปรอีกครั้ง ซึ่งจะคล้ายกับภาษาแบบตรรกะ

ภาษาสำหรับการเขียนโปรแกรมสมัยใหม่นั้นควรจะสนับสนุนข้อมูลในรูปแบบนามธรรมในบางรูปแบบ ซึ่งทุกโปรแกรมจะได้รับประโยชน์จากการแปลงสภาพของข้อมูลนี้ให้เป็นหน่วยของการทำงานที่เหมาะสม โดยโปรแกรมที่มีขนาดใหญ่นั้นจะถูกจัดการด้วยวิธีในลักษณะนี้ โดยการที่ขาดการจัดการข้อมูลในรูปแบบนามธรรมที่ดีจะทำให้อาจเกิดข้อบกพร่องที่ร้ายแรงต่อภาษานั้นๆ ได้

แน่นอนว่าการนำงานที่มีอยู่แล้วกลับมาใช้ซ้ำนั้นเป็นหนึ่งในเป้าหมายที่สำคัญของงานทางด้านวิศวกรรมซอฟต์แวร์ โดยการนำงานที่มีอยู่แล้วกลับมาใช้นั้นจะช่วยลดรูปแบบที่ไม่จำเป็นในการจัดการด้านชนิดของข้อมูล ดังนั้นสำหรับงานทางด้านวิศวกรรมนั้นภาษานั้นๆ จะต้องสนับสนุนการทำงานในลักษณะพ้องรูป (polymorphism) ซึ่งแน่นอนว่าลักษณะเหล่านี้จะต้องเกี่ยวข้องกับภาษาแบบ imperative ภาษาเชิงวัตถุ และภาษาเชิงลักษณะงาน โดยการออกแบบภาษานั้นต้องคำนึงถึงปัจจัยดังต่อไปนี้

2.2.1 ความเรียบง่าย (Simplicity)

ความเรียบง่ายนั้นควรจะเป็นเป้าหมายของการออกแบบภาษา โดยภาษาสำหรับการเขียนโปรแกรมนั้นคือเครื่องพื้นฐานของโปรแกรมเมอร์ ดังนั้นโปรแกรมเมอร์จะต้องมีความเข้าใจภาษาเหล่านั้นอย่างลึกซึ้ง แน่นอนว่าภาษาสำหรับการเขียนโปรแกรมนั้นจะต้องช่วยโปรแกรมเมอร์ในการแก้ไขปัญหา โดยตัวภาษาเองก็ควรจะสนับสนุนให้โปรแกรมเมอร์สามารถแสดงวิธีแบบธรรมชาติหรือปกติในการแก้ปัญหา ซึ่งแน่นอนว่ามันควรจะช่วยให้เราค้นพบวิธีการแก้ปัญหานั้นๆ ตั้งแต่แรกโดยภาษาที่มีความซับซ้อนมากๆ นั้นจะก่อให้เกิดปัญหาที่ยากต่อการจัดการและควบคุม ดังนั้นภาษาที่มีขนาดใหญ่และซับซ้อนนั้นไม่จำเป็นจะต้องใช้สำหรับการแก้ปัญหาที่มีขนาดใหญ่ทุกครั้งไป แต่ขึ้นอยู่กับความเหมาะสมของภาษาที่จะดำเนินการปัญหาที่เกิดตามมา

ภาษาปาสคาล (Pascal programming language) นั้นถูกออกแบบมาโดยมีวัตถุประสงค์ที่จำกัด ซึ่งจริงๆ แล้วค่อนข้างประสบความสำเร็จในด้านของการทำงาน เนื่องจากว่าภาษาปาสคาลนั้นมีขนาดเล็ก แต่มีลักษณะการออกแบบภาษาที่เหมาะสมซึ่งง่ายต่อการทำความเข้าใจและบริหารจัดการในการแก้ไขปัญหา ซึ่งเพียงพอในการตอบสนองต่อการแก้ปัญหาที่หลากหลาย [13]

อย่างไรก็ตามมักจะเกิดปัญหาระหว่างเป้าหมายที่เรียบง่ายกับความต้องการที่แท้จริงของภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไป เพราะว่าภาษาสำหรับการพัฒนาโปรแกรมแบบทั่วไปนั้นจะต้องสามารถตอบสนองต่อการแก้ปัญหาที่หลากหลาย รวมทั้งปริมาณงานทางด้านขนานที่จะถูกนำมาสร้างเป็นโปรแกรมขนาดใหญ่โดยจะประกอบไปด้วยโปรแกรมย่อยหลายๆ ส่วนซึ่งจะถูกพัฒนาจากโปรแกรมเมอร์จำนวนมากและจะถูกนำกลับมาใช้ซ้ำในภายหลัง ซึ่งภาษาในลักษณะนี้อาจจะต้องมีการทำงานที่ซับซ้อนมากกว่าภาษาปาสคาล

วิธีการที่เหมาะสมที่สุดสำหรับแก้ปัญหานั้นทำให้เกิดความขัดแย้งกันเองระหว่างความเรียบง่ายของภาษากับความต้องการทำงานของภาษา สำหรับการพัฒนาโปรแกรมแบบทั่วไปนั้นจะต้องสามารถทำการแบ่งลักษณะต่างๆ ของภาษาออกจากกัน ซึ่งจะทำให้โปรแกรมเมอร์แต่ละคนสามารถจัดการปัญหาเหล่านั้นได้เฉพาะในส่วนที่จำเป็น ซึ่งวิธีการนี้ผู้ออกแบบภาษาจะต้องหลีกเลี่ยงการสร้างหรือมีปฏิสัมพันธ์ใดๆ ที่คาดไม่ถึงซึ่งอาจจะทำให้เกิดจากความไม่เข้ากันของภาษาในบางส่วน ซึ่งอาจนำไปสู่ความเข้าใจผิดและทำให้โปรแกรมเมอร์ไม่สามารถทำความเข้าใจกับภาษานั้นๆ ได้

ภาษา PL/I ทำให้เกิดข้อวิพากษ์วิจารณ์อย่างมาก สำหรับความผิดพลาดในการแก้ไขปัญหาและควบคุมการทำงานที่ซับซ้อนที่เกิดขึ้นจากลักษณะตัวภาษาของมันเอง ยกตัวอย่างเช่นการบวกเลข 25 + 1/3 ซึ่งภาษา PL/I นั้นต้องถูกเขียนในลักษณะคำสั่งทางคณิตศาสตร์ที่ดูซับซ้อนเกินไปแต่ให้ผลลัพธ์ที่เท่ากันคือ 5.3! โดยบางครั้งการทำงานนี้อาจมีการตัดทอนตัวเลขที่มีนัยยะสำคัญออกไป ดังนั้นโปรแกรมเมอร์จะต้องมีความระมัดระวังในการดำเนินงานทางด้านคณิตศาสตร์เพราะอาจจะเกิดข้อผิดพลาดที่ไม่ตั้งใจขึ้นได้ [14]

ภาษา ADA เองก็ถูกวิพากษ์วิจารณ์สำหรับขนาดและความซับซ้อนของมันเช่นกัน อย่างไรก็ตามภาษา ADA เองก็สามารถจัดการและควบคุมความซับซ้อนของภาษาได้อย่างดี ซึ่งถ้าโปรแกรมเมอร์ของภาษาปาสคาลมาทดลองเขียนโปรแกรมภาษา ADA นั้นโปแกรมสามารถทำงานได้เป็นปกติ แต่ด้วยความไม่คุ้นเคยของโปรแกรมเมอร์อาจจะละเลยการจัดการข้อผิด

ของโปรแกรม โดยการจัดการข้อผิดพลาดนั้นระบบจะช่วยแสดงข้อความออกมาเพื่อบอกถึงข้อผิดพลาด เพื่อปัญหาเหล่านั้นจะได้ถูกจัดการอย่างเหมาะสม

แม้กระทั่งภาษา Huskell เองที่เป็นภาษาที่ไม่มีความซับซ้อน แต่ก็ยังคงมีจุดที่ก่อให้เกิดความยุ่งยากกับโปรแกรมเมอร์เช่นเดียวกัน โดยการอนุมานตัวแปรประเภทที่พ้องรูปกัน (Polymorphic) ซึ่งจะมีความเป็นไปได้ว่ามีการอนุมานถึงฟังก์ชั่นที่แตกต่างจากฟังก์ชั่นที่ตั้งใจจะเรียกใช้งานจริง ซึ่งถ้าเกิดขึ้นในส่วนของโปรแกรมขนาดใหญ่จะทำให้ยากต่อการตรวจสอบ ดังนั้นเพื่อหลีกเลี่ยงความสับสนทำให้โปรแกรมเมอร์ต้องระบุประเภทของตัวแปรและฟังก์ชั่นในทุกๆ ที่ เป็นเหตุให้ข้อมูลมีความซับซ้อนมากขึ้น แต่ว่าการเขียนในลักษณะนี้จะช่วยให้ง่านต่อการทำความเข้าใจซึ่งความซับซ้อนในบางครั้งนั้นก็มักจะเป็นสิ่งที่ดีในการออกแบบภาษา [15, 16]

การพ้องรูป (Polymorphism) การทำ Method overloading และการบีบบังคับ (Coercions) นั้นมีประโยชน์ในตัวเอง แต่การนำสิ่งต่างๆ มารวมกันในภาษาเดียวอาจนำไปสู่การมีปฏิสัมพันธ์ที่ไม่คาดคิด [17, 18] เช่น ภาษา C++ และ ภาษา Java เมื่อมีการรวมตัวเข้าด้วยกันของ overloading และหลักการสืบทอด (Inheritance) ดังตัวอย่างที่เกิดขึ้นในคลาสและซับคลาสของ Java ดังรูปที่ 2.1

```
class Line {
      private int length;
      (1) public void set (int newLength) {
            length = newLength;
      }
      . . .
}
class ColoredLine extends Line {
      public static final RED = 0, GREEN = 1, BLUE = 2;
            private int color;
            (2) public void set (int newColor) {
            color = newColor;
      }
. . .
}
```

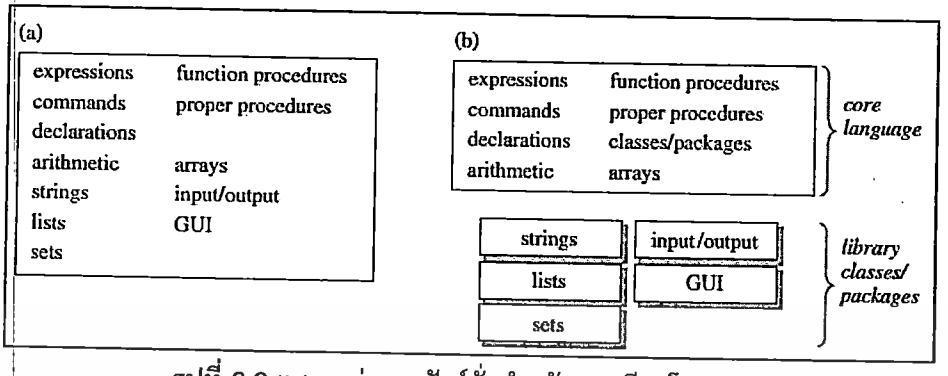
รูปที่ 2.1 แสดงการพ้องรูปของภาษา Java

ในมุมมองของโปรแกรมเมอร์ตัวแปร length และ color นั้นเป็นตัวแปรที่มีชนิดที่แตกต่างกัน ดังนั้นเมธอด (2) ควรเรียกใช้งาน เมธอด (1) แต่ในความเป็นจริงนั้นส่วนการรับค่าทั้งสองเมธอดนั้นเป็นเลขจำนวนเต็มเหมือนกัน ดังนั้นแทนที่จะเป็นการเรียกใช้งานแต่แท้จริงแล้วกลับเป็นการสืบทอดการทำงานของคลาสแทน

เมื่อกลับมายังรูปแบบของการที่พยายามจะควบคุมความซับซ้อน นักออกแบบภาษาสำหรับเขียนโปรแกรมนั้นไม่ควรคาดการณ์สิ่งอำนวยความสะดวกต่างๆ สำหรับความต้องการให้โปรแกรมเมอร์ ซึ่งแทนที่จะสร้างสิ่งอำนวยความสะดวกต่างๆ ให้โปรแกรมเมอร์ใช้ควรจะให้โปรแกรมเมอร์เขียนโปรแกรมเพื่อกำหนดสิ่งอำนวยความสะดวกต่างๆ ที่พวกเขาต้องการด้วยตัวเอง

ด้วยข้อเสนอดังกล่าวทำให้ภาษาหลักมีขนาดที่เล็กลงแต่เต็มไปด้วยกลไกลที่มีประสิทธิภาพซึ่งเต็มไปด้วยส่วนต่างๆ ที่สามารถนำกลับมาใช้ซ้ำได้ ดังนั้นภาษาควรเตรียมชนิดของตัวแปรพื้นฐานที่

มีขนาดเล็ก และตัวแปรที่สามารถทำงานร่วมกันได้ ซึ่งจะทำให้สามารถนำมาผสมเพื่อสร้างตัวแปรชนิดใหม่ๆ ออกมาใช้งานได้เมื่อจัดการรูปแบบอย่างเหมาะสม



รูปที่ 2.2 แสดงกลุ่มของฟังก์ชั่นสำหรับการเขียนโปรแกรม

โดยกลุ่มของฟังก์ชั่นต่างๆ จะประกอบไปด้วยตัวแปรประเภท สตริง ลิสต์ และอินพุท/เอาท์พุท ซึ่งจำเป็นมากสำหรับการเขียนโปรแกรมโดยสามารถสรุปได้ดังรูปที่ 2.2 ซึ่งจะทำให้เกิดผลกระทบต่อเนื่องจากภาษาจนถึงกลุ่มของฟังก์ชั่น นอกจากนี้โปรแกรมเมอร์เองสามารถสร้างและกำหนดฟังก์ชั่นต่างๆ ขึ้นมาเรียกใช้งานเองได้

โดยทางภาษา Java เองได้สนับสนุนลักษณะที่กล่าวมาข้างต้นเป็นอย่างดี โดยแกนของภาษา Java นั้นมีความเรียบง่ายจึงทำให้ภาษา Java ง่ายตัวการทำความเข้าใจ โดยข้อมูลของภาษา Java ถูกจัดเก็บอยู่ในรูปของคลาสและรวมลักษณะต่างๆ ที่เหมือนกันไว้ในแพ็คเกจเดียวกันซึ่งทำให้สามารถสร้างโปรแกรมออกมาได้อย่างเหมาะสม [19]

2.2.2 ประสิทธิภาพ (Efficiency)

แน่นอนว่าการทำงานของคอมพิวเตอร์ในยุคแรกๆ นั้นจะช้ามากเมื่อเทียบกับปัจจุบันนี้ นอกจากจะทำงานช้าแล้วยังมีพื้นที่สำหรับจัดเก็บข้อมูลที่ค่อนข้างจำกัดดังนั้นภาษาในยุคแรกๆ อย่างภาษา Fortran และภาษา Cobol จึงได้ถูกออกภายให้ทำงานภายใต้สภาวะข้อจำกัดเหล่านี้เพื่อที่จะช่วยให้คอมพิวเตอร์สามารถทำงานได้อย่างเต็มประสิทธิภาพ แต่ในระยะหลังมีการพัฒนาเพื่อเพิ่มประสิทธิภาพการทำงานของ ทำให้คอมพิวเตอร์สามารถทำงานได้อย่างรวดเร็วและสามารถจัดเก็บข้อมูลได้เป็นจำนวนมาก ซึ่งทำให้ตัวแปลภาษาสามารถสร้างโค้ดออกมาได้อย่างมีประสิทธิภาพ

โดยตอนนี้เข้าใจว่าการหลักการทำงานที่จะทำให้งานนั้นมีประสิทธิภาพสูงสุดนั้นค่อนข้างเป็นไปได้ยาก เนื่องจากโดยปกติแล้วส่วนหลักของการทำงานประมาณ 90% จะอยู่ในช่วงการประมวลข้อมูลต่างๆ และจะมีแค่ 10% เท่านั้นที่ถูกใช้ไปในส่วนการทำงานในส่วนของโค้ด ดังนั้นจึงเป็นเรื่องยากเมื่อต้องทำการเพิ่มประสิทธิภาพของงานในส่วนของ 10% มาช่วยดึงงานในส่วนของ 90% ที่ไม่สามารถเพิ่มประสิทธิภาพของงานได้ อย่างไรก็ตามภาษาทุกๆ ภาษาควรจะมีประสิทธิภาพที่เหมาะสมสำหรับการพัฒนาโปรแกรม โดยความเหมาะสมนั้นต้องขึ้นอยู่กับวัตถุประสงค์ของแต่ละภาษาเช่น ภาษาที่มีไว้สำหรับการเขียนโปรแกรมที่เกี่ยวข้อง

กับงานทางด้านระบบจะต้องมีประสิทธิภาพสูงกว่าภาษาที่มีไว้สำหรับการเขียนโปรแกรมประยุกต์เป็นต้น

โดยทั่วไปแล้วประสิทธิภาพการทำงานของภาษาจะได้รับอิทธิพลจากแนวคิดเป็นอย่างมาก ซึ่งบางแนวคิดอย่างเช่น ชนิดตัวแปรแบบพลวัตร (Dynamic-typing) การพ้องรูป (Polymorphism) การทำงานเชิงวัตถุ (Object orientation) การจัดการเนื้อที่ที่ไม่ได้ถูกเรียกใช้แบบอัตโนมัติ (Automatic deallocation, Garbage collector) [4, 19] นั้นจะมีต้นทุนการทำดำเนินการที่สูงแทบทั้งสิ้น การเขียนโปรแกรมเชิงตรรกะนั้นจริงๆ แล้วจะมีประสิทธิภาพที่ด้อยกว่าการเขียนโปรแกรมเชิงฟังก์ชั่น ในทางกลับกันการเขียนโปรแกรมเชิงฟังก์ชั่นกลับมีประสิทธิภาพที่ด้อยกว่าแบบ imperative ดังนั้นนักออกแบบภาษาจำเป็นต้องตัดสินใจว่าประโยชน์ของแต่ละแนวคิดนั้นมีข้อดี ข้อเสียอย่างไร

2.2.3 ไวยากรณ์ (Syntax)

หลักเกณฑ์ที่สำคัญมากสำหรับไวยากรณ์ของภาษาสำหรับการเขียนโปรแกรมคือต้องสามารถอ่านและเข้าใจได้ง่าย ซึ่งเป็นเพราะว่าโปรแกรมนั้นถูกเขียนแค่ครั้งเดียวแต่จะถูกอ่านซ้ำไปซ้ำมาหลายครั้งจากโปรแกรมเมอร์คนอื่นๆ ดังนั้นนักออกแบบภาษาควรต้องเลือกไวยากรณ์ที่อนุญาตและส่งเสริมให้โปรแกรมเมอร์สามารถเขียนโปรแกรมได้อย่างเหมาะสม สามารถอ่านได้ง่ายโดยโปรแกรมเมอร์คนอื่น โดยโปรแกรมเมอร์นั้นควรจะต้องมีอิสระในการกำหนดรูปแบบและความหมายของโปรแกรม โดยคำเฉพาะหรือคำสำคัญนั้นควรจะต้องเลือกสัญลักษณ์ที่คุ้นเคย ซึ่งจะเป็นเรื่องง่ายสำหรับการเขียนโปรแกรม

หลักเกณฑ์ที่สำคัญอีกอันหนึ่งคือโค้ดนั้นจะต้องไม่สามารถแก้ไขความหมายได้ง่ายเนื่องจากความผิดพลาดเล็กน้อย ซึ่งตัวอย่างของความผิดพลาดที่รุนแรงนั้นเกิดขึ้นในภาษา Fortran คือการประกาศตัวแปร DO1I=1,25 โดยความตั้งใจจริงจะให้เป็นการวนซ้ำจาก 1 ถึง 25 แต่เกิดข้อผิดพลาดทำให้ประกาศเป็น DO1I=1.25 ดังนั้นทำให้โปรแกรมเกิดข้อผิดพลาดเนื่องจากความไม่ตั้งใจอันเกิดจากลักษณะภาษาได้ [20]

ไวยากรณ์ของภาษาควรจะใช้สัญลักษณ์ทางคณิตศาสตร์ที่คุ้นเคยให้มากเท่าที่จะเป็นไปได้อย่างเช่น ในคณิตศาสตร์เครื่องหมาย = นั้นมีความหมายว่า “เท่ากับ” ซึ่งในคำจำกัดความของภาษา Huskell และ ADA เครื่องหมายนี้ถูกใช้สำหรับความเท่าเทียมกันซึ่งจะตรงกับความหมายทางคณิตศาสตร์ แต่สำหรับภาษาซี และภาษา Java นั้นหมายความว่า “จะเท่ากับ” ซึ่งเป็นการกำหนดค่า โดยจะไม่สอดคล้องกับความหมายทางคณิตศาสตร์ ซึ่งแน่นอนว่าการตีความหมายในลักษณะนี้ได้ส่งผลให้เกิดข้อผิดพลาดในภาษาซีขึ้นคือ

```
if (x = y) printf(". . .");
```

คำสั่งนี้ต้องการที่จะพิมพ์ข้อความบางอย่างออกมา โดยถ้าพบว่าค่าที่ตัวแปร x และตัวแปร y นั้นมีค่าเท่ากัน ซึ่งในความเป็นจริงคำสั่งนี้เป็นการกำหนดค่าจากตัวแปร y ให้ตัวแปร x ซึ่งแน่นอนว่าข้อผิดพลาดที่เกิดจากไวยากรณ์นั้นทำให้โปรแกรมทำงานผิดพลาดได้

นอกจากนี้ส่วนที่ลึกของไวยากรณ์ควรจะมีความโปร่งใส โดยควรจะมีรูปแบบไวยากรณ์ที่มีความชัดเจนจารูปแบบของความหมาย ซึ่งรูปแบบไวยากรณ์และความหมายใดๆ นั้นควรจะตอบสนองต่องานใดงานหนึ่งเท่านั้น

โดยภาษาซี และ C++ นั้นแสดงให้เห็นถึงส่วนที่มีปัญหาในการออกแบบไวยากรณที่ไม่ค่อยจะเหมาะสม โดยใช้คำจำกัดความของรูปแบบที่แตกต่างกันถึง 4 รูปแบบคือ enum, struct, typedef และ union ซึ่งจะทำให้การประกาศตัวแปรนั้นเกิดความสับสนขึ้นได้ง่าย อย่างเช่น

```
char * x, y;

char s, t[];
```

โดยการประกาศตัวแปร x ชนิด pointer แต่ในส่วนของตัวแปร y นั้นเป็นตัวแปรประเภทตัวอักษรธรรมดาเท่านั้น เช่นเดียวกับตัวแปร s แต่ในส่วนของตัวแปร t นั้นกลับเป็นการประกาศตัวแปรชนิดอาร์เรย์ โดยจะตรงกันข้ามภาษา Java ซึ่งจะทำการประกาศตัวแปรและพารามิเตอร์ต่างๆ ไว้ในที่ๆ กันเดียว ตัวอย่างเช่น

```
char[] s, t;
```

ซึ่งหมายความว่าทั้งตัวแปร s และ t ต่างถูกประกาศเป็นตัวแปรชนิดอาร์เรย์ของตัวอักษรธรรมดาเหมือนๆ กัน

รูปแบบของไวยากรณ์ที่เหมือนกันไม่ควรจะอ้างอิงถึงงานในบริบทที่แตกต่างกัน อย่างเช่น ในรูปประโยคของ `(char)x` ซึ่งในบริบทหนึ่งนั้นหมายถึงการประกาศตัวแปรที่ชื่อ x และมีชนิดของตัวแปรเป็นตัวอักษร แต่ในอีกบริบทหนึ่งนั้นเป็นการกำหนดค่าของตัวแปร x ให้มีลักษณะเป็นตัวอักษร โดยตัวอย่างภาษาที่มีความซับซ้อนที่ทำให้เกิดความสับสนของไวยากรณ์นั้นมาจากภาษา C++ โดยสมมติว่าคลาส Widget นั้นมีฟังก์ชั่น 2 ตัวโดยตัวแรกทำการรับค่าของเลขจำนวนเต็มสำหรับเป็นค่าตั้งต้น และอีกตัวหนึ่งไม่มีการกำหนดค่าเริ่มต้น ซึ่งสามารถสร้างและกำหนดค่าสำหรับวัตถุทั้ง 2 ตัวดังนี้

```
Widget* w1 = new Widget(7); // calls the first constructor

Widget* w2 = new Widget(); // calls the second constructor
```

แต่ถ้าต้องการสร้างวัตถุแบบพื้นฐาน 2 ชนิดซึ่งสามารถจองพื้นที่บนหน่วยความจำจะต้องดำเนินการดังนี้

```
Widget w3(7); // calls the first constructor

Widget w4(); // intended to call the second constructor!
```

จะพบว่าการประกาศตัวแปร w3 นั้นสามารถทำงานได้เป็นปกติ แต่การประกาศตัวแปร w4 นั้นจะไม่สามารถทำงานได้ เนื่องจากการประกาศในลักษณะนี้จะถูกมองว่าเป็นการประกาศฟังก์ชั่นไม่ใช่การประกาศตัวแปร

โดยโปรแกรมนั้นเป็นโครงสร้างหลักของการกำหนดความหมายว่าการทำงานในขั้นตอนต่างๆ นั้นจะต้องดำเนินไปในลักษณะใด ส่วนไวยากรณ์นั้นเป็นสัญลักษณ์ซึ่งโปรแกรมเมอร์เลือกแนวคิดและความหมายเพื่อไปใช้ในการแก้ปัญหา ซึ่งไวยากรณ์ของภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์นั้นเป็นส่วนที่สำคัญนั้นจะต้องง่ายต่อการเรียนรู้และความเข้าใจ เพื่อที่นำไปพัฒนาโปรแกรมได้อย่างตรงความต้องการ

2.2.4 ช่วงชีวิตของภาษา (Language life cycle s)

ภาษาใหม่ทุกภาษาต้องผ่านขั้นตอนหลายส่วนในการออกแบบก่อนจะมีการนำมาใช้งานจริงโดยโปรแกรมเมอร์โดยประกอบไปด้วยขั้นตอนต่างๆ ดังนี้ [1]

- มีการกำหนดข้อกำหนดและจุดประสงค์ต่างๆ ของภาษา
- มีการออกแบบโครงสร้างไวยากรณ์และความหมายของภาษา โดยแนวคิดและความหมายของภาษานั้นจะต้องถูกกำหนดไว้อย่างชัดเจนตั้งแต่แรก เพื่อเลือกวิธีการสร้างภาษาให้มีเอกลักษณ์และเรียบง่าย เมื่อได้ความหมายที่ต้องการแล้วหลังจากนั้นจึงค่อยทำการออกไวยากรณ์ของภาษา
- มาตรฐานของภาษานั้นจะต้องถูกออกแบบและเตรียมการ โดยมาตรฐานของภาษานั้นจำเป็นต้องมีลักษณะดังนี้ จะต้องเขียนและทำความเข้าใจง่าย จะต้องมีความสมบูรณ์และสอดคล้องกันของภาษา จะต้องโปร่งใสปราศจากความกำกวมของภาษา เพื่อทำให้โปรแกรมเมอร์สามารถวิจารณ์ภาษาใหม่ได้อย่างเหมาะสม
- ใช้มาตรฐานของภาษาเพื่อการเริ่มต้นสำหรับการสร้างภาษาใหม่ เพื่อให้โปรแกรมเมอร์สามารถได้รับประสบการณ์จากการใช้ภาษาใหม่
- จะต้องมีมาตรฐานสำหรับการพัฒนาภาษา ซึ่งจะทำให้โปรแกรมเมอร์สามารถออกความคิดเห็นถึงผู้ออกแบบภาษา ซึ่งจะได้นำข้อเสนอแนะไปออกแบบและปรับปรุงภาษาให้ดียิ่งขึ้น
- ถ้าภาษาใหม่นั้นได้รับการยอมรับจะทำให้เกิดการปรับปรุงการวิธีการพัฒนาภาษา และจะมีหนังสือต่างๆ เพื่ออธิบายการทำงานและหลักการของภาษา อีกทั้งชุมชนของโปรแกรมเมอร์ที่ใช้ภาษาในลักษณะเดียวกันก็จะเกิดขึ้น โดยสิ่งที่ตามมาอีกหลังจากนั้นคือคือหลักสูตรที่เป็นสากลซึ่งแน่นอนว่าภาษาเหล่านั้นยังคงได้รับการแก้ไข และปรับปรุงอยู่เสมอ

2.3 ตัวแปลภาษา (Compiler)

ตัวแปลภาษาเป็นโปรแกรมแปลภาษาที่ทำหน้าที่แปลภาษาระดับสูงมาเป็นภาษาเป้าหมาย ซึ่งอาจหมายถึง ภาษาเครื่อง (Assembly Language) โดยที่ระหว่างขั้นตอนการทำการแปลภาษาเกิดมีข้อผิดพลาดเกิดขึ้นตัวแปลภาษาจะหยุดการทำงานเพื่อให้ผู้ใช้งานทำการแก้ไขข้อผิดพลาด แต่ถ้าไม่มีข้อผิดพลาดตัวแปลภาษาจะทำงานต่อไปจนจบ และเกิดระบบที่ต้องการขึ้นเพื่อจะได้ทำการติดต่อสื่อสาร และใช้งานกับเครื่องคอมพิวเตอร์ต่อไป

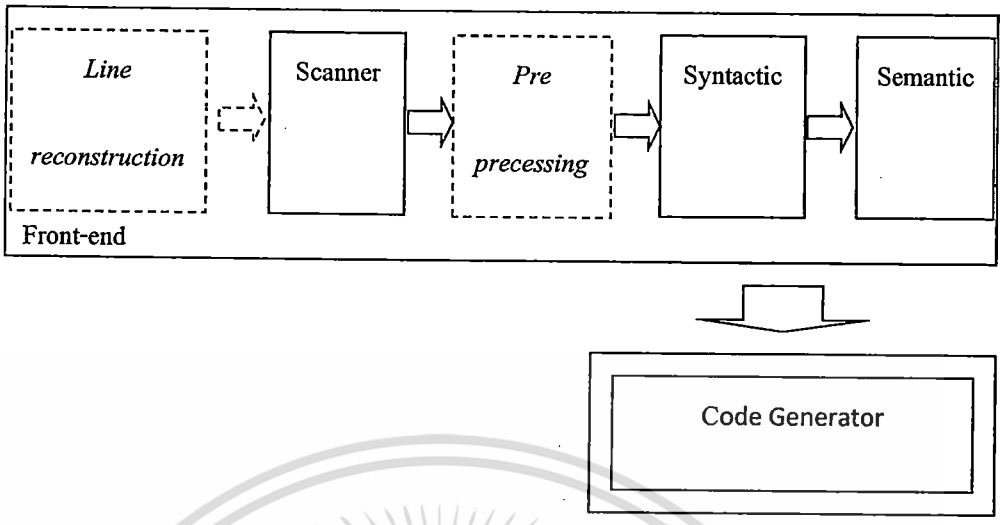
เนื่องจากคอมพิวเตอร์จะเข้าใจแต่ภาษาเครื่อง (Machine Language) ซึ่งมีลักษณะเป็นชุดของข้อมูลที่สร้างขึ้นจากรหัสของระบบเลขฐานสอง (เลข 0 และ 1 เท่านั้น) โดยจะหมายถึงสถานะของ

ไฟฟ้าที่มีสองสถานะ คือ เปิดและปิด ดังนั้นวิธีการควบคุมการทำงานของเครื่องด้วยวิธีการดังกล่าวค่อนข้างจะยุ่งยาก และซับซ้อนเนื่องจากโปรแกรมเมอร์จะต้องจำรหัสคำสั่งซึ่งเข้าใจได้ยาก ทำให้การตรวจสอบหาข้อผิดพลาดของโปรแกรมนั้นยากขึ้นไปอีก ดังนั้นจึงต้องอาศัยโปรแกรมที่มีความสามารถในการแปลภาษาคอมพิวเตอร์(Translator) ในการแปลภาษาคอมพิวเตอร์ภาษาต่างๆไปเป็นภาษาเครื่อง โดยโปรแกรมที่แปลจากโปรแกรมต้นฉบับแล้วจะเรียกว่า “Object Code” ซึ่งจะประกอบด้วย รหัสคำสั่งที่คอมพิวเตอร์สามารถเข้าใจ และนำไปปฏิบัติต่อได้ [10, 11] โดยทั่วไปจะแยกชนิดของตัวแปลภาษาออกเป็นลักษณะดังนี้

- ตัวแปลภาษา (Compiler) : เป็นตัวแปลภาษาระดับสูง อย่างเช่น ภาษาปาสคาล ภาษา Cobol และภาษา Fortran ให้เป็นภาษาเครื่อง โดยการทำงานจะใช้หลักการแปลโปรแกรมต้นฉบับทั้งโปรแกรม โดยจะเรียกใช้งานข้อมูลที่ถูกบันทึกไว้ในลักษณะของแฟ้มข้อมูลหรือไฟล์ โดยเมื่อมีความต้องการที่จะเรียกใช้งานโปรแกรมอีกก็จะสามารถเรียกจากไฟล์ได้โดยตรงโดยไม่ต้องทำการแปลอีกทำให้การทำงานเป็นไปอย่างรวดเร็ว ขณะที่คอมไพล์โปรแกรมต้นฉบับที่เขียนขึ้นด้วยภาษาระดับสูง ตัวแปลภาษาจะตรวจสอบโครงสร้างไวยากรณ์ที่คำสั่งและข้อมูลที่จะใช้ในการคำนวณ และเปรียบเทียบ จากนั้นตัวแปลภาษาจะสร้างรายงานเพื่อสรุปที่อาจเกิดข้อผิดพลาดของโปรแกรมนั้นๆ โดยข้อมูลเหล่านั้นมีประโยชน์ในการช่วยโปรแกรมเมอร์ในการแก้ไขโปรแกรมให้ทำงานได้อย่างถูกต้อง
- ล่าม (Interpreter) : เป็นตัวแปลระดับสูงเช่นเดียวกับตัวแปลภาษา แต่จะการแปลคำสั่งพร้อมกับทำงานไปด้วย โดยจะทำการแปลคำสั่งทีละคำสั่งไปเรื่อยๆ จนหมดทั้งโปรแกรม ทำให้การแก้ไขโปรแกรมกระทำได้ง่ายและรวดเร็ว การแปลโดยใช้ล่ามจะไม่สร้างไฟล์โปรแกรมที่จะถูกเรียกใช้งาน ดังนั้นจะต้องมีการแปลใหม่ทุกๆ ครั้งที่มีการเรียกใช้งาน ตัวอย่างภาษาที่ใช้ล่ามเป็นตัวแปล เช่น ภาษาเบสิก (BASIC)
- แอสเซมเบลอ (Assembler) : เป็นตัวแปลภาษาแอสเซมบลี (Assembly language) โดยจะทำการแปลภาษาจาก Object Code ให้ไปเป็นชุดคำสั่งที่ใช้ควบคุมการทำงานของเครื่องซึ่งเรียกว่า (Operation Code: OpCode) โดย OpCode ที่ได้นั้นจะต้องถูกแปลภาษาให้เหมาะสมกับสถาปัตยกรรมของเครื่องแต่ละชนิด

2.3.1 โครงสร้างของตัวแปลภาษา (Structure of compiler)

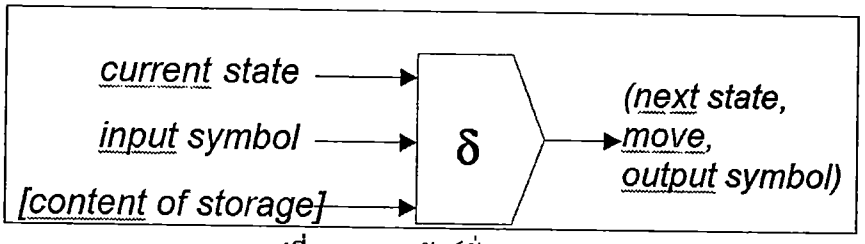
ตัวแปลภาษานั้นเปรียบเสมือนตัวเชื่อมระหว่างภาษาระดับสูงกับฮาร์ดแวร์ และคอมพิวเตอร์ โดยตัวแปลภาษานั้นจะมีหน้าที่ตรวจสอบความถูกต้องของรหัสต้นฉบับ โดยวิธีที่จะทำการตรวจสอบรหัสต้นฉบับและทำการแปลออกมาได้อย่างถูกต้องและมีประสิทธิภาพนั้น ตัวแปลภาษาต้องประกอบไปด้วยโครงสร้างหลัก 2 ส่วนหลักดังรูปที่ 2.3 คือ ส่วนหน้า (Front end) และส่วนหลัง (Back end) โดยแต่ละส่วนนั้นถูกออกแบบมาเพื่อทำงานในลักษณะที่แตกต่างกัน



รูปที่ 2.3 โครงสร้างของตัวแปลภาษา

โดยที่ตัวแปลภาษาส่วนหน้านั้นจะมีหน้าที่ทำการวิเคราะห์เพื่อสร้างโค้ดรูปแบบภายในของระบบขึ้นมาซึ่งเรียกว่า รูปแบบการแทนในระยะกลาง (Intermediate Representation, IR) อีกทั้งยังเป็นส่วนที่มีการจัดการตารางสัญลักษณ์ ซึ่งตารางสัญลักษณ์นี้จะเป็นโครงสร้างข้อมูลที่โยงไปสู่ความหมายที่โค้ดต้นแบบนั้นเกี่ยวข้องอยู่ซึ่งได้แก่ ตำแหน่ง ชนิด และขอบเขต ซึ่งทั้งหมดนี้ถูกดำเนินการในขั้นตอนต่างๆที่อยู่ภายในตัวแปลภาษาส่วนหน้า [10, 11] โดยตัวแปลภาษาส่วนหน้านั้นจะประกอบไปด้วยส่วนต่างๆ ดังต่อไปนี้

- Line reconstruction คือส่วนที่สำหรับจัดการภาษาที่อนุญาตให้ใช้ชุดของอักขระที่มีลักษณะเหมือนกับคำเฉพาะ ซึ่งโดยส่วนมากภาษาสำหรับการเขียนโปรแกรมสมัยใหม่จะไม่อนุญาตให้ทำ ทำให้ไม่มีส่วนนี้ในตัวแปลภาษา
- ส่วนของการวิเคราะห์คำ (Lexical analysis) ในส่วนนี้จะทำการแยกอักขระออกเป็นชิ้นๆ โดยเรียกว่าลักษณะเฉพาะหรือเครื่องหมายซึ่งลักษณะเฉพาะแต่ละตัวคือหน่วยย่อยที่ถูกกำหนดขึ้นเพื่อใช้ในภาษาเหล่านั้น โดยจะเป็นตัวแทนของคำสำคัญ คำเฉพาะ คำที่ถูกใช้ระบุชื่อ และเครื่องหมายและสัญลักษณ์ต่างๆ ซึ่งโครงสร้างทางไวยากรณ์ของลักษณะเฉพาะเหล่านี้จะถูกจัดอยู่ในรูปแบบของภาษาปกติ (Regular language) ซึ่งสถานะจำกัดดังรูปที่ 2.4 โดยจะถูกสร้างขึ้นมาเพื่อจดจำนิพจน์ปกติ (Regular expression) และดำเนินการกับความหมายเหล่านี้ โดยทั่วไปในส่วนนี้จะถูกเรียกว่าตัววิเคราะห์คำ (Lexical analyzer หรือ Scanner)



รูปที่ 2.4 แสดงฟังก์ชั่น Transition

- ส่วนก่อนการประมวลผล (Preprocessing) ภาษาบางชนิดอย่างเช่นภาษา C นั้น ต้องการส่วนก่อนการประมวลผลสำหรับสนับสนุนการแทนตำแหน่งของมาโคร (Macro) และทำการแปลภาษาอย่างมีเงื่อนไข ซึ่งโดยทั่วไปส่วนก่อนการประมวลผลนี้จะอยู่ก่อนส่วนของการตรวจสอบไวยากรณ์และความหมายของภาษาซึ่งในกรณีของภาษา C นั้น ในส่วนของส่วนก่อนการประมวลผลนี้จะจัดการสิ่งที่ใช้เป็นสัญลักษณ์ ได้ดีกว่าส่วนของการตรวจสอบไวยากรณ์ แต่อย่างไรก็ตามภาษาบางชนิดสามารถจัดการรูปแบบการแทนตำแหน่งของมาโครในลักษณะของการตรวจสอบไวยากรณ์ได้ด้วยเช่นกัน
- ส่วนของการตรวจสอบไวยากรณ์ (Syntactic analysis) ส่วนของการตรวจสอบไวยากรณ์นี้ได้นำเอาสิ่งที่ใช้เป็นสัญลักษณ์ซึ่งได้จากส่วนของการวิเคราะห์คำมาทำการตรวจสอบรูปแบบว่าเข้ากับรูปแบบที่ได้ถูกระบุให้อยู่ในระบบภาษาหรือไม่ โดยในส่วนนี้จะสร้างโครงสร้างต้นไม้เพื่อทำการแทนค่าชุดของลักษณะเฉพาะ เพื่อทำการตรวจสอบโครงสร้างต้นไม้นั้นต้องถูกต้องตามกฎของไวยากรณ์ที่เน้นรูปนัย (Formal grammar) ที่ได้ถูกระบุอยู่ในไวยากรณ์ของภาษาหรือไม่
- ส่วนตรวจสอบความหมายของภาษา (Semantic analysis) ในส่วนนี้จะทำการตรวจสอบความหมายเพื่อความถูกต้องของภาษา อย่างเช่นการตรวจสอบชนิดของข้อมูล ซึ่งส่วนตรวจสอบความหมายของภาษานี้เป็นส่วนสุดท้ายก่อนนำไปสู่การแปลงรหัสต้นฉบับไปเป็นภาษาเครื่อง

ตัวแปลภาษาส่วนหลังโดยบางทีอาจจะมีความสับสนกับส่วนของตัวสร้างรหัสเครื่อง (Code generator) เพราะว่ามีการทำงานที่คาบเกี่ยวกันระหว่างส่วนที่ทำหน้าที่แปลงรหัสต้นฉบับไปเป็นรหัสแอสเซมบลีกับส่วนของการปรับปรุงประสิทธิภาพการทำงานของรหัสเครื่องในตัวแปลภาษา ซึ่งตัวแปลภาษาบางชนิดอาจจะมีการสร้างตัวแปลภาษาส่วนกลาง (Middle end) ขึ้นเพื่อมาทำหน้าที่ในการจัดการโค้ดให้มีการทำงานได้อย่างมีประสิทธิภาพ (Code optimization) โดยจะแยกส่วนนี้ออกมาจากการสร้างรหัสเครื่องของตัวแปลภาษา ซึ่งสามารถแบ่งลักษณะการทำงานของส่วนต่างๆของตัวแปลภาษาออกมาได้เป็น 3 ส่วนดังนี้

- การวิเคราะห์ (Analysis) ส่วนนี้เป็นส่วนที่จะรวบรวมข้อมูลทั้งหมดจากรูปแบบการแทนในระยะกลางซึ่งจะถูกแปลงมาจากรหัสต้นฉบับ โดยทั่วไปการวิเคราะห์นี้เปรียบเสมือนการวิเคราะห์ลักษณะของการเคลื่อนไหวของข้อมูลเพื่อจะนำไปสร้าง use-define chains, dependence analysis, alias analysis, pointer analysis, escape analysis เป็นต้น ซึ่งการวิเคราะห์ที่ถูกต้องและแม่นยำนั้นจะเป็นขั้นตอนพื้นฐานของการเพิ่มประสิทธิภาพของการทำงานด้านตัวแปลภาษา

ซึ่งแผนภาพการเรียกใช้งาน (Call graph) และแผนภาพควบคุมการทำงาน (Control flow graph) จะต้องถูกสร้างที่ส่วนนี้ด้วย

- ส่วนปรับปรุงประสิทธิภาพ (Optimization) ในส่วนนี้รูปแบบการแทนในระยะกลางที่ได้จากส่วนข้างต้นนี้จะถูกแปลงให้อยู่ในรูปแบบที่สามารถทำงานได้อย่างรวดเร็วยิ่งขึ้น โดยใช้วิธีการจัดการต่างๆ เช่น inline expansion, dead code elimination, constant propagation, loop transformation, register allocation และ automatic parallelization

ส่วนการแปลงรหัสให้เป็นภาษาเครื่อง (Code generation) ในส่วนนี้รูปแบบการแทนในระยะกลางจะถูกจัดการให้ออกไปเป็นภาษาเหมาะสมที่เครื่องคอมพิวเตอร์จะสามารถนำไปใช้ต่อได้ โดยทั่วไปแล้วมักจะอยู่ในรูปของภาษาเครื่อง (Machine language) ซึ่งในส่วนนี้ได้มีการจัดการเกี่ยวกับทรัพยากร และการจัดเก็บข้อมูลของเครื่องด้วย อย่างเช่น ดูว่าตัวแปรที่กำหนดไว้นั้นต้องเหมาะสมกับขนาดของหน่วยความจำ และการทำงานขั้นตอนต่างๆนั้นต้องเข้ากันกับคำสั่งเครื่องตามที่ได้ระบุไว้

2.3.2 การสร้างตัวแปลภาษา (An implementation of compiler)

ในทฤษฎีการสร้างตัวแปลภาษานั้นขั้นตอนแรกต้องมีการกำหนดนิยามของภาษาก่อน โดยหลังจากนิยามความหมายของภาษาได้แล้ว ขั้นตอนถัดมาต้องทำการสร้างตัวแปลภาษาตามลำดับขั้นที่ได้แสดงไว้ในส่วนของโครงสร้างของตัวแปลภาษา โดยจะต้องทำการสร้างส่วนของการวิเคราะห์คำ ขึ้นมาก่อนและภายในส่วนของการวิเคราะห์คำนี้จะต้องนิยามความหมายและคำสำคัญของภาษาดังนี้

2.3.2.1 ส่วนการวิเคราะห์คำ

ในส่วนนี้จะทำการแยกอักขระออกเป็นชิ้นๆ ซึ่งเรียกว่าลักษณะเฉพาะ หรือเครื่องหมาย โดยลักษณะเฉพาะของชุดอักขระแต่ละตัวจะถูกนิยามเป็นหน่วยย่อยที่เพื่อระบุสำหรับใช้ในภาษาเหล่านั้น โดยจะเป็นตัวแทนของคำสำคัญหรือคำเฉพาะ คำที่ถูกใช้ระบุชื่อ และ ตัวดำเนินการและสัญลักษณ์ต่างๆ โดยสามารถแบ่งการสร้างออกเป็นส่วนๆ ได้แก่

ไวยากรณ์ศัพท์ (Lexical grammar) โครงสร้างของภาษานั้นจะถูกกำหนดด้วยกลุ่มของกฎซึ่งใช้ระบุในการสร้างส่วนตัวคัดแยกคำ ซึ่งกฎที่ถูกกำหนดขึ้นนั้นจะเป็นรูปแบบของภาษาปกติ โดยจะนำตัวอักษรและอักขระทั้งหมดที่ถูกนิยามในภาษามาใช้เป็นตัวระบุในการสร้างสิ่งที่ใช้เป็นสัญลักษณ์ หรือเครื่องหมาย (Token) ซึ่งเป็นชุดของสตริงที่อ้างอิงจากกฎโดยเป็นเหมือนสัญลักษณ์ หรือเครื่องหมาย (คำที่ถูกใช้ระบุชื่อ, ตัวเลข และ เครื่องหมายคอมมา เป็นต้น) ซึ่งกระบวนการสร้างชุดของลักษณะเฉพาะจะเรียกว่าการตัดคำจากข้อความ (Tokenization) ซึ่งลักษณะเฉพาะนั้นจะเป็นอะไรก็ได้ตามที่ถูกกำหนดให้มีความหมายในภาษา ยกตัวอย่างนิพจน์ของภาษาซี

```
sum = 3+2;
```

โดยสามารถแบ่งออกมาเป็นลักษณะเฉพาะได้ดังตารางที่ 2.1

ตารางที่ 2.1 แสดงการตัดคำของกระบวนการตัดคำจากข้อความ

คำศัพท์ (Lexeme)	ชนิดของศัพท์ (Token type)
sum	ตัวอักษร
=	เครื่องหมาย
3	ตัวเลข
+	เครื่องหมาย
2	เครื่องหมาย
;	เครื่องหมายปิดประโยค

โดยลักษณะเฉพาะจะถูกกำหนดโดยนิพจน์ปกตินั้นจะถูกจัดการโดยตัวคัดแยกคำ อย่างเช่น Lex ซึ่งตัวคัดแยกคำที่เป็นเครื่องมืออย่าง Lex นั้นจะอ่านชุดของตัวอักษร แล้วทำการจัดให้อยู่ในกลุ่มที่ได้ถูกระบุไว้ในภาษาเรียกว่ากระบวนการคัดแยกคำ

2.3.2.2 ส่วนการตรวจสอบไวยากรณ์

จากส่วนที่ผ่านมาตัวสร้างลักษณะเฉพาะ (Token generation) หรือส่วนของการวิเคราะห์คำนั้นได้ทำการตัดชุดของอักขระจากรหัสต้นแบบที่นำเข้ามาให้อยู่ในรูปแบบของ นิพจน์ปกติที่ได้ทำการระบุไว้ ซึ่งข้อมูลดังกล่าวจะไหลเข้าสู่กระบวนการถัดไปที่เรียกว่าส่วนของการตรวจสอบไวยากรณ์

โดยในส่วนนี้จะใช้ไวยากรณ์ไม่พึ่งบริบท (Context-free grammar) เป็นตัวที่สร้างภาษาไม่พึ่งบริบทจะมีลักษณะ $A \longrightarrow \Psi$ โดยที่ A เป็นสัญลักษณ์ไม่ท้ายสุด และ Ψ จะเป็นสัญลักษณ์ท้ายสุดหรือไม่ท้ายสุดก็ได้ ภาษาไม่พึ่งบริบทเป็นพื้นฐานทางทฤษฎีของวากยสัมพันธ์ของภาษาโปรแกรมส่วนใหญ่ [21, 22]

อย่างไรก็ตามไม่ใช่ว่าไวยากรณ์ทุกตัวต้องถูกกำหนดด้วยไวยากรณ์ไม่พึ่งบริบท เท่านั้นยังมีลักษณะบางชนิด อย่างเช่น การตรวจสอบชนิดของตัวแปร (Type validity) และการกำหนดสำหรับลักษณะสำหรับประกาศตัวแปร ซึ่งบางครั้งอาจจะถูกกำหนดโดยไวยากรณ์ข้อมูล (Attribute Grammars, AG) [23] โดยการระบุโครงสร้างไวยากรณ์ของภาษาด้วยวิธีการทั้ง 2 ลักษณะวิธีนั้นถูกออกแบบมาเพื่อการทำงานที่แตกต่างกันดังต่อนี้

- ไวยากรณ์ไม่พึ่งบริบท เป็นวิธีการจัดการไวยากรณ์ซึ่งใช้สำหรับสร้างภาษา โดยสามารถจัดให้อยู่ในรูปแบบที่เป็นระเบียบภายใต้เงื่อนไขที่รัดกุม โดยที่ไวยากรณ์ไม่พึ่งบริบทนั้นถูกแสดงออกมาให้อยู่ในรูปของโครงสร้างลำดับชั้นของชอมสกี้ในระดับที่ 2 ซึ่งในเทอมของกฎการสร้างทุกๆ กฎจะถูกจัดให้อยู่ในรูปแบบ

$$A \longrightarrow \Psi$$

โดยที่ A คือเป็นสัญลักษณ์ไม่ท้ายสุดและ Ψ จะเป็นสัญลักษณ์ท้ายสุดหรือไม่ท้ายสุดก็ได้ โดยการเขียนกฎในลักษณะนี้สามารถนำไปเขียนในรูปของโครงสร้างต้นไม้ได้เช่นเดียวกัน โดยที่กำหนดให้สัญลักษณ์ไม่ท้ายสุดนั้นเป็นโหนดอยู่บนสุดซึ่งสามารถมีโหนดลูกได้ ส่วนสัญลักษณ์ท้ายสุดหรือไม่ท้ายสุดจะถูกกำหนดให้อยู่ในส่วนของโหนดที่ไม่มีลูกหรือโหนดอื่นต่อ ซึ่งไวยากรณ์ไม่พึ่งบริบทนั้นสามารถทำหน้าที่เป็นตัวกลางสำหรับอธิบาย และออกแบบภาษาสำหรับการเขียนโปรแกรม และตัวแปลภาษาได้โดยจะถูกใช้สำหรับวิเคราะห์และตรวจสอบความถูกต้องของไวยากรณ์ของภาษาปกติ

- ไวยากรณ์ข้อมูล คือกฎเกณฑ์สำหรับการกำหนดลักษณะข้อมูลสำหรับการสร้างรูปแบบของไวยากรณ์ที่เน้นรูปนัย โดยการทำงานจะเกิดขึ้นกับโหนดของโครงสร้างต้นไม้นามธรรม (Abstract Syntax Tree, AST) [24] เมื่อภาษานั้นถูกดำเนินการโดยตัวแปลภาษา ซึ่งลักษณะข้อมูลนั้นสามารถแบ่งออกได้เป็น 2 ชนิด คือข้อมูลที่ถูกสังเคราะห์ (Synthesized attributes) และข้อมูลที่ถูกสืบทอด (Inherited attributes) โดยที่ถูกสังเคราะห์นั้นเป็นผลลัพธ์ที่เกิดจากข้อที่ได้จากการประมวลผลตามกฎ ซึ่งบางครั้งจะสามารถใช้ค่าจาก ข้อมูลที่ถูกสืบทอดได้ส่วนข้อมูลที่ถูกสืบทอดนั้นจะใช้ค่าที่ได้จากโหนดแม่ (Parent node) [23]

2.4 วิธีการแปลงโปรแกรมและระบบสำหรับสร้างตัวแปลภาษา (Program Transformation Techniques and Systems for compiler construction)

นอกจากการอธิบายทฤษฎีการสร้างแปลภาษาแล้ว เทคนิคและวิธีการแปลงโปรแกรม โครงร่างและระบบนั้นจำเป็นต้องกล่าวถึง เพื่อให้เข้าใจถึงทางเลือกสำหรับวิธีที่ใช้ดำเนินการในการแก้ปัญหา โดยในหัวข้อนี้วิธีการต่างๆ จะถูกนำเสนอ อย่างเช่น การทำส่วนขยายภาษาที่คล้ายกัน (Assimilating language extensions) โครงร่างของตัวแปลภาษาเปิด (Open compiler framework) การเปลี่ยนรูปของโค้ดตัวกลาง (Intermediate code transformation) รวมไปถึง ภาษาเฉพาะทาง (DSLs) ด้วยเช่นกัน ซึ่งตัวอย่างของระบบทั้งหมดนี้จะถูกจัดกลุ่มซึ่งสามารถจะถูกจัดอยู่ในกลุ่มที่มากกว่าหนึ่งกลุ่มใด

การเปลี่ยนแปลงโปรแกรมเป็นขั้นตอนที่โปรแกรมเกิดการเปลี่ยนแปลงปรับปรุง และแก้ไขเพื่อวัตถุประสงค์บางอย่าง เช่นการเพิ่มหรือขยายประสิทธิภาพ การมุ่งเน้นขยายบางส่วนของโปรแกรมนั้นเพื่อที่จะขยายไวยากรณ์ของภาษาให้รองรับงานบางอย่างซึ่งทำให้เกิดรูปแบบการทำงานที่สะดวกมากยิ่งขึ้น เป็นต้น เพื่อให้บรรลุเป้าที่ข้างต้น วิธีการต่างๆ จะต้องถูกนำมาใช้ในการสร้างและ

พัฒนา โดยในส่วนนี้วิธีการต่างๆ จะถูกกล่าวถึงเพื่อมุ่งเน้นให้ถึงวิธีการที่ช่วยสำหรับการปรับปรุงเปลี่ยนแปลงโปรแกรมรวมไปถึงการสร้างระบบ

2.4.1 การทำส่วนขยายภาษาที่คล้ายกัน (Assimilating language extensions)

วิธีการนี้เป็นวิธีการที่ถูกนำมาใช้บ่อยครั้งในการเปลี่ยนแปลงการทำงานของระบบ โดยจะมีการเขียนโครงสร้างของภาษาใหม่บนรากภาษาเดิมที่มีมาตรฐานอยู่แล้วซึ่งจะถูกรู้จักเป็นอย่างดีว่าเป็นการขยายจากข้อมูลตัวหนึ่งมาเป็นข้อมูลอีกตัวหนึ่ง

ประโยชน์ที่ได้จากการขยายภาษาที่คล้ายกันนี้นั้นคือภาษาใหม่นั้นจะถูกออกแบบและพัฒนาบนภาษาระดับสูง ซึ่งสามารถทำให้ภาษาใหม่มีความหมายที่ง่ายต่อการทำความเข้าใจ แต่ว่าข้อด้อยของวิธีในลักษณะนี้คือบางโครงสร้างของภาษาใหม่จะไม่สามารถทำงานเข้ากันกับภาษาเดิม ซึ่งทำให้ต้องใช้ความพยายามในการออกแบบและพัฒนาภาษาอย่างมาก ดังนั้นบางครั้งอาจจะทำให้ไม่สามารถพัฒนาและล้มเลิกการพัฒนาภาษานั้นๆ ได้

โครงร่าง Spoon (Spoon framework) [25] และ OpenJava (บางทีเรียกว่า OJ) คือ 2 โปรแกรมสำหรับพัฒนาภาษา Java [26] ที่ถูกจัดลักษณะวิธีการทำงานอยู่ในกลุ่มนี้ ซึ่งโครงร่างระบบของทั้ง 2 ระบบนี้อาศัยการสะท้อนช่วงการแปลภาษา (Compile-time reflection) [27] และสร้างโครงสร้างต้นไม้นามธรรมเพื่อสำหรับสร้างการเปลี่ยนรูป โดยทั้ง 2 วิธีการรับประกันความถูกต้อง แต่ปัญหาของวิธีการ ทั้ง 2 ชนิดคือบางทีโครงสร้างของภาษาใหม่ไม่สามารถทำงานเข้ากันกับภาษาเดิมดังได้กล่าวมาแล้ว

2.4.2 โครงร่างของตัวแปลภาษาเปิด (Open compiler framework)

ตัวแปลภาษาของภาษาสำหรับการเขียนโปรแกรมนั้นเป็นตัวเลือกที่เหมาะสมเมื่อการขยายโครงสร้างการทำงานของภาษาการเขียนโปรแกรมใหม่ แต่ปัญหาที่กลายเป็นปัญหาที่สำคัญในการขยายโครสร้างภาษาใหม่ด้วยของวิธีการนี้คือถึงแม้ว่าจะมีการเปลี่ยนไวยากรณ์เพียงเล็กน้อยก็อาจจะส่งผลถึงโครงสร้างส่วนใหญ่ที่ไม่ได้รับการพัฒนาด้วย นอกจากนี้จะต้องมีความรู้ความเข้าใจอย่างละเอียดลึกซึ้งเกี่ยวกับการทำงานของตัวแปลภาษาซึ่งอาจจะเกินความจำเป็นเมื่อแค่ต้องการขยายโครงสร้างภาษาเพียงเล็กน้อย

อีกวิธีการที่ช่วยในการแก้ไขปัญหาในการสร้างตัวแปลภาษาคือการใช้ตัวแปลภาษาที่มีอยู่แล้วโดยกการใช้ตัวแปลภาษาส่วนหลังนั้นเพื่อสร้างรหัสเป้าหมาย และใช้โครงร่างของตัวแปลภาษาที่สามารถขยายตัวแปลภาษาส่วนหน้า โดยโครงร่างของตัวแปลภาษาเปิดนั้นใช้สำหรับสร้างและขยายภาษา โดยมีการเตรียมลักษณะวิธีการที่เหมาะสมทำให้ง่ายต่อการออกแบบและขยายภาษา

Polyglot เป็นเครื่องมืออีกชนิดหนึ่งที่เป็นตัวอย่างของโครงร่างของตัวแปลภาษาเปิด [28] โดยJastAddJ [29] และ Dryad [30] นั้นเป็นตัวอย่างของการขยายความสามารถของตัวแปลภาษาแบบสมบูรณ์ โดย Polyglot นั้นจะดำเนินการโดยใช้ประโยชน์ของการขยายงานตัวเองซึ่งจะทำให้ Polyglot มีความสามารถในการแปลภาษาที่เพิ่มมากกขึ้น แต่ข้อเสียของวิธีการในลักษณะนี้คือระบบจะถูกจำกัดการทำงานของการขยายภาษาต้องสามารถพกพาด้วยภาษา Java ไปด้วยในขณะที่ JastAddJ และ Dryad เป็นตัวแปลภาษาที่ทำงานได้สมบูรณ์ แต่

ปัญหาคือต้องการความรู้ความเข้าใจที่ถูกต้องของภาษาใดๆ ที่ต้องการพัฒนา อีกสิ่งหนึ่งที่เป็นปัญหาที่สำคัญของเครื่องมือในลักษณะนี้คือจะต้องทำให้ภาษานั้นมีรูปแบบที่ทันสมัยอยู่เสมอ โดยเฉพาะเมื่อมีการเปลี่ยนรูปแบบของภาษา Java

2.4.3 การเปลี่ยนรูปของโค้ดตัวกลาง (Intermediate code transformation)

บางวิธีการของการแปลงโปรแกรมนั้นจะมุ่งเน้นไปที่ส่วนของการขยายการทำงานของตัวแปลภาษาส่วนหน้า เนื่องจากว่าการขยายการทำงานในส่วนหลังของตัวแปลภาษานั้นค่อนข้างซับซ้อนและทำให้เกิดความผิดพลาดได้ง่าย ซึ่งเมื่อการขยายการทำงานของตัวแปลภาษาส่วนหน้ามาถึงข้อจำกัดดังนั้นงานวิจัยจึงได้มีการมุ่งเน้นในส่วนของการแปลงรหัสกลางซึ่งอยู่ในตัวแปลภาษา Java และภาษา C# ซึ่งวิธีนี้เกินกว่าการทำงานของส่วนหลังของตัวแปลภาษา ซึ่งทำให้เกิดความเป็นไปได้สำหรับการเปลี่ยนแปลงและปรับปรุงรหัสกลาง

แต่ข้อด้อยของวิธีการในลักษณะนี้คือการจัดการนั้นต้องทำอยู่ในระดับล่างซึ่งจะมีความซับซ้อนสูงทำให้ยากในการดำเนินการ แต่จะได้ประโยชน์มากเพราะสามารถจัดการรหัสได้อย่างเหมาะสมกับงานและสถาปัตยกรรม

เมื่อมองดูลักษณะงานในภาษา Java ซึ่งจะมีโครงร่าง 2 ประเภทสามารถอธิบายการทำงานในลักษณะนี้ โดยเริ่มแรกจะมีการเตรียมฟังก์ชั่นส่วนที่ใช้สำหรับขยายภาษาซึ่งจะอยู่ในรูปของภาษาระดับสูงอย่างภาษา Java โดยทั้ง 2 ประเภทคือ ASM [31] และ BCEL [32] โดยทั้ง ASM [31, 33] และ BCEL นี้จะทำการจัดการคลาสของภาษา Java โดยระบบนั้นจะเตรียมข้อมูลประเภทคำสั่งของไบท์โค้ด เพื่อที่จะให้โปรแกรมเมอร์มุ่งเน้นไปในการแปลงรูปข้อมูลแทนที่จะจัดการคำสั่งของไบท์โค้ด ซึ่งความซับซ้อนของการพัฒนาในลักษณะนี้นั้นคุ้มค่าพอที่จะแลกเพื่อที่จะปรับปรุงงานการทำงานของไบต์โค้ด

2.4.3.1 โครงร่าง ASM (ASM framework)

ASM คือโครงร่างสำหรับจัดการไบต์โค้ดโดยถูกออกแบบเพื่อสร้างและจัดการไบต์โค้ดแบบพลวัตร ASM [33] นั้นไม่ได้มาจากตัวย่อหรือความหมายใดๆ แต่อ้างอิงมาจากคำสำคัญ asm ของภาษา C ซึ่งอนุญาตให้บางฟังก์ชั่นของทำงานจัดการภาษาเครื่องได้ โดยแนวคิดหลักของ ASM คือต้องสามารถถูกนำมาสร้างระบบให้เร็วที่สุดเท่าที่เป็นไปได้นั้น และจะไม่สนับสนุนการใช้ตัวแทนแบบคลาสวัตถุ โดยจะใช้รูปแบบการเป็นผู้เยี่ยมชม (Visitor pattern) แต่จะไม่ถูกเยี่ยมชมโครงร่างต้นไม้ด้วยวิธีการเชิงวัตถุ โดยรูปแบบการเป็นผู้เยี่ยมชมนั้นจะสามารถจัดรูปแบบการทำงานของโค้ดที่เฉพาะเจาะจงและรูปแบบอื่นๆ โดยทั่วไปได้ซึ่งจะไม่ทำการสร้างวัตถุหนึ่งวัตถุสำหรับทำงานคำสั่งโค้ด 1 โค้ด โดย ASM นั้นจะมุ่งเน้นความสำคัญของการทำงานไปที่การเปลี่ยนรูปโค้ดเป็นหลักแทนการจัดการคำสั่งด้วยภาษาระดับล่างที่ยุ่งยากและซับซ้อน

2.4.3.2 โครงร่าง BCEL (BCEL framework)

โครงร่าง BCEL (Byte Code Engineering Library) คือโครงร่างที่ใช้วิเคราะห์ สร้าง และจัดการไบต์โค้ดของภาษา Java [32, 35] ซึ่งโครงสร้างภายในของคลาสแต่ละคลาสนั้นจะประกอบไปด้วยข้อมูลที่เป็นรูปของสัญลักษณ์ต่างๆ ที่จะถูกคิดค้นขึ้นเพื่อใช้สำหรับดำเนินการโดยตัวแปลภาษา เช่น ฟังก์ชั่นการทำงาน ตัวแปร คำสั่งของไบต์โค้ด

ซึ่งวัตถุแต่ละตัวจะสามารถถูกอ่านได้จากไฟล์ที่มีอยู่แล้วโดยโปรแกรม และทำการเขียนข้อมูลทั้งหมดอีกครั้ง โดยส่วนการทำงานที่น่าสนใจคือการสร้างคลาสใหม่ตั้งแต่ต้นของการทำงาน โดยชุดคำสั่งการทำงานของ BCEL นั้นถูกใช้ในงานหลายๆ อย่างเช่น

- งานด้านแปลภาษา Java แบบย้อนกลับ
- งานด้านการสร้างความหมายของภาษาใหม่ อย่างเช่น วิธีเชิงลักษณะ (Aspect-oriented) [34] ถูกพัฒนาขึ้นโดยใช้โครงร่างของ BECL เป็นตัวแยกส่วนโครงสร้างของคลาสจากการนิยามจุดตัด (Point-cut) และในส่วนของการสร้างโครงสร้างคลาสที่ถูกจัดการโดยวิธีเชิงลักษณะเป็นต้น

2.4.4 ภาษาเฉพาะทาง (DSLs)

อย่างที่ได้กล่าวมาข้างต้นบ้างแล้วว่าเป็นภาษาขนาดเล็กที่ถูกสร้างขึ้นสำหรับงานเฉพาะทางบางประเภท โดย Van Deursen [36] ได้รวบรวมงานต่างๆ ที่เกี่ยวกับภาษาเฉพาะทาง โดยนิยามของภาษาเฉพาะทางทั้งหลายนั้นคือหน่วยภาษาขนาดเล็กที่สามารถทำงานภายใต้ข้อจำกัดทางเครื่องหมายและรูปแบบที่เป็นนามธรรม แต่มุมมองอื่นๆ ที่มีต่อภาษาเฉพาะทางนั้นคือเป็นภาษาที่เป็นภาษาที่จำกัดมากกว่าเป็นภาษาสำหรับการเขียนโปรแกรม

ภาษาเฉพาะทางในบริบทของวิธีการแปลงโปรแกรมนั้นอนุญาตให้ระบุความสามารถการทำงานของภาษาใดๆ โดยจะถูกแปลงโดยโปรแกรม ซึ่งแน่นอนว่าสามารถถูกจัดการให้อยู่ในรูปแบบ 2 ชนิดคือรูปแบบภายนอก (External DSLs) และรูปแบบภายใน (Internal DSLs) ซึ่งบางครั้งอาจถูกเรียกว่ารูปแบบภาษาเฉพาะทางที่ถูกฝังติด (Embedded DSLs) แต่รูปแบบภายนอกที่จะถูกพูดถึงต่อไปนี้มีความสามารถในการจัดการให้มีความใกล้เคียงกับภาษาเฉพาะทางที่เป็นต้นแบบ หรือใกล้เคียงกับรูปแบบเดิม โดยรูปแบบจะถูกจำกัดโดยความสามารถของนักออกแบบภาษาในการสร้างตัวแปลงที่จะสามารถวิเคราะห์คำในประโยคและสร้างโปรแกรมที่สามารถทำงานได้แบบปกติกับภาษาราก ซึ่งแตกต่างจากรูปแบบภายในที่ใช้แนวคิดที่จะสืบทอดโครงสร้างพื้นฐานจากภาษาสำหรับการเขียนโปรแกรมนั้นๆ ทำให้สามารถทำงานได้อย่างเหมาะสมกับภาษาแต่ละภาษา โดยภาษาจะถูกพัฒนาโดยการระบุความหมายบนโครงสร้างของภาษาหลัก (Host language) ซึ่งจะนำไปสู่การสร้างภาษาใหม่

ส่วนลักษณะของรูปแบบภายนอกนั้นจะสามารถนิยามความหมายภาษาได้อย่างเต็มรูปแบบซึ่งทำให้สามารถสนับสนุนรูปแบบปกติได้เป็นอย่างดี โดยข้อจำกัดนั้นจะขึ้นอยู่กับความสามารถในการสร้างตัววิเคราะห์ภาษาที่สามารถทำงานได้อย่างเป็นปกติบนภาษานั้นๆ

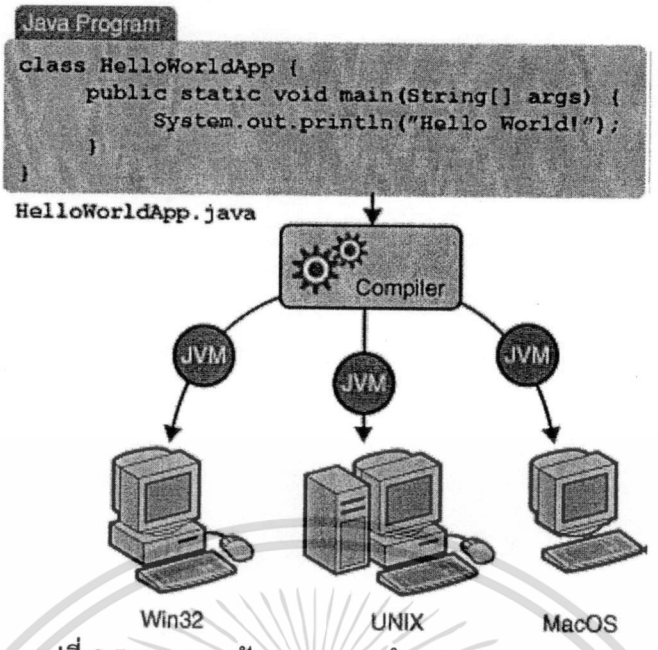
ในทางกลับกันรูปแบบภายในนั้นซึ่งสามารถทำงานบนโครงสร้างภาษาเดิมได้แต่จะไม่สามารถจัดการลักษณะงานเฉพาะทางได้ด้วยรูปแบบของไวยากรณ์ที่เหมาะสมกับปัญหาเหล่านั้นได้ ซึ่งอาจจะทำให้เกิดข้อผิดพลาดจากการส่งข้อมูลที่คลาดเคลื่อนได้ ทางเลือกสำหรับภาษาเฉพาะทางนั้น ในบริบทของโปรแกรมที่ใช้ดำเนินการพัฒนานั้นจะขึ้นอยู่กับความต้องการของการแปลงข้อมูลพื้นฐานเพื่อสร้างโค้ดที่เหมาะสม โดย Stratego/XT [37] นั้นเป็นโปรแกรมสำหรับสร้างโค้ดข้อมูลโดยทำงานด้วยภาษา Stratego ซึ่งเป็นตัวอย่างที่ดีสำหรับงานทางด้านนี้ โดยใช้ความสามารถของภาษา Stratego และเครื่องมือต่างๆ อย่างเหมาะสม

2.5 ภาษา Java (Java Programming Language)

ภาษา Java นั้นเป็นภาษาสำหรับการเขียนโปรแกรมเชิงวัตถุซึ่งพัฒนาโดย Sun Microsystems ภาษาJava เป็นภาษาที่มีความเหมาะสมในการเขียนโปรแกรมภาษาวัตถุประสงค์ทั่วไปภาษาหนึ่ง ซึ่งโปรแกรมของภาษา Java สามารถทำงานในได้อยู่ในแพลตฟอร์มที่หลากหลาย เนื่องจากว่า Java มี Java Virtual Machine (JVM) ที่สามารถทำงานได้อยู่บนหลากหลายสถาปัตยกรรม (ดังรูปที่ 2.5)

โปรแกรมของภาษา Java สามารถทำงานได้อย่างรวดเร็วและมีประสิทธิภาพเท่าที่โปรแกรมภาษา C++ เนื่องจากการดำเนินงานของ JVM โดยจะเพิ่มไปลักษณะการทำงานแบบภาษา C++ เข้าไปในส่วนของการจัดการหน่วยความจำ และนอกจากนี้ก็ยังขยายความสามารถของภาษาให้การสนับสนุนสำหรับการทำงานแบบมัลติเธรด

ภาษา Java มีมาตรฐานของภาษา (Java Language Specification, JSL) [19] ที่ถูกระบุไว้อย่างดี โดยนำมารวมกับความสามารถในการทำงานบนหลากหลากแพลตฟอร์มของ JVM และความสามารถในด้านสนับสนุนการขยายโครงสร้างภาษา Java นี้เองทำให้ภาษาสำหรับการเขียนโปรแกรมสมัยใหม่หลายภาษาถูกขยายความสามารถออกมาจากภาษา Java โดยการขยายความสามารถของภาษา Java นั้นทาง Java เองได้มีการอนุญาตให้ทำการขยายได้อยู่ใน 2 ลักษณะ คือ การขยายความสามารถแบบภายใน (Internal extended) และ การขยายความสามารถแบบภายนอก (External extended)



รูปที่ 2.5 แสดงสถาปัตยกรรมการทำงานของภาษา Java

2.5.1 การขยายความสามารถภาษา Java แบบภายใน (Internal extended Java programming language)

แนวคิดของการขยายความสามารถการทำงานของภาษา Java นั้นได้ถูกนักออกแบบภาษาหลายคนประยุกต์และออกแบบจนเป็นภาษาใหม่แล้วใยหลายๆ ภาษา เพื่อตอบสนองต่อความต้องการในความสามารถของการทำงานที่หลากหลาย การขยายความสามารถแบบภายในนั้นจะสามารถทำได้จากโครงสร้างของภาษา Java ที่สนับสนุนโดยตรงด้วยการสร้างคลาสหรือเฟรมเวิร์คขึ้นมาเพื่อรองรับการทำงาน อย่างเช่น ภาษา JRuby ซึ่งเป็นการนำภาษา Java มาเป็นล่ามของภาษา Ruby ซึ่งจะสนับสนุนเอกลักษณ์การทำงานของภาษา Ruby เช่น ตัวแปรแบบพลวัตร (Dynamic-typing) แต่การขยายความสามารถเพื่อสนับสนุนการทำงานของภาษาอื่นในลักษณะนี้จะต้องคำนึงด้วยไวยากรณ์ของภาษา Java เป็นสำคัญ และเช่นเดียวกันภาษา Jython [38] ซึ่งสนับสนุนเอกลักษณ์การทำงานของภาษา Python [39] จะต้องคำนึงด้วยไวยากรณ์ของภาษาด้วยเช่นกัน

2.5.2 การขยายความสามารถภาษา Java แบบภายใน (External extended Java programming language)

การขยายความสามารถของภาษาแบบภายนอกนี้จะตรงกันข้ามกับวิธีการขยายความสามารถแบบภายใน โดยการขยายความสามารถของภาษาในลักษณะนี้จะเข้าไปทำการแก้ไขในตัวไวยากรณ์ของภาษาโดยตรงซึ่งจะส่งผลให้โครงสร้างของภาษาเกิดการเปลี่ยนแปลง แน่นอนว่าโดยทั่วไปแล้วจะอ้างอิงการขยายความสามารถของภาษา Java จากมาตรฐานของภาษา Java ที่ได้ถูกกำหนดไว้ก่อนหน้า แต่บางครั้งการแก้ไขอาจจะมีผลกระทบต่อโครงสร้าง

ของภาษา การขยายความสามารถแบบภายนอกนี้สามารถตอบสนองต่อการทำงานที่เพิ่มเติมขึ้นโดยตรงและง่ายกว่าการขยายความสามารถแบบภายใน เพราะโครงสร้างและไวยากรณ์จะสนับสนุนอย่างเต็มที่ อย่างเช่น ภาษา JR ซึ่งภาษา JR เป็นภาษาที่ขยายออกมาจากภาษาเพื่อตอบสนองต่อการทำงานทางด้านการทำงานแบบคู่ขนาน (Concurrency) ซึ่งใช้ลักษณะของภาษาจากภาษา SR [40] มาจัดการทางด้านนี้ โดยการขยายความสามารถของภาษาในลักษณะนี้ทำให้ภาษา JR จะต้องมีคำเฉพาะ (Reserve word) เพิ่มขึ้นมาในคำสั่งของภาษาโดยตรง เป็นต้น

แม้ว่าการขยายความสามารถแบบภายในจะสามารถทำได้โดยง่ายและรวดเร็ว แต่ข้อจำกัดของการขยายความสามารถของภาษาในลักษณะนี้คือจะต้องทำงานภายใต้โครงสร้างไวยากรณ์ของภาษาหลักนั่นคือภาษา Java ซึ่งจะส่งผลให้ไม่ได้รับความสะดวกในการจัดการโครงสร้างของโค้ดที่จะต้องอยู่ในลักษณะที่แตกต่างจากภาษาที่ถูกขยายลงไปค่อนข้างมาก ซึ่งจะตรงกันข้ามกับวิธีการขยายความสามารถแบบภายนอกที่โครงสร้างและไวยากรณ์จะสนับสนุนอย่างเต็มที่ทำให้สามารถจัดการโครงสร้างของโค้ดได้อย่างสะดวก แต่ว่าการขยายความสามารถของภาษาในลักษณะภายนอกนี้จะใช้เวลาและทำได้ยากว่าแบบภายในมาก

2.6 ลักษณะคลาสของภาษา Java (The Java class file format)

โค้ดของภาษา Java ที่จะถูกดำเนินการโดย JVM นั้นจะต้องถูกแปลงให้เป็นโค้ดกลางเพื่อใช้กับเครื่อง ระบบปฏิบัติการโดยจะอยู่ในลักษณะเลขฐาน 2 ที่เป็นอิสระ (Independent binary format) โดยไฟล์ที่ถูกแปลแล้วนั้นจะถูกเก็บอยู่ในรูปที่เรียกว่าคลาสไฟล์ (.class) ซึ่งคลาสไฟล์จะถูกระบุอยู่ในรูปที่เป็นตัวแทนของคลาส หรืออินเตอร์เฟสโดยจะประกอบไปด้วยรายละเอียดต่างๆ เช่น การเรียงลำดับของไบต์โค้ดสำหรับการทำงานต่างๆ [42]

โดยคลาสนั้นจะประกอบไปด้วยข้อมูล 8 บิตไบท์ซึ่งถ้าอยู่ในรูปแบบที่เป็น 16 บิต 32 บิต และ 64 บิตจะถูกอ่านด้วยลักษณะของ 2, 4, 8 ตามลำดับ โดยใช้ลักษณะข้อมูล 8 บิตไบท์ ซึ่งขนาดของข้อมูลจะถูกเก็บเป็นลำดับตามขนาดของข้อมูล

โครงสร้างของคลาสที่ถูกแปลแล้วนั้นจะอยู่ในรูปที่เรียบง่าย แตกต่างจากการแปลข้อมูลแบบปกติโครงสร้างนี้จะประกอบไปด้วยข้อมูลที่โดยมากแล้วเป็นสัญลักษณ์ที่มาจากโค้ดต้นฉบับดังนี้ (ตาราง 2.2)

ตารางที่ 2.2 แสดงโครงสร้างข้อมูลของ Java

<table>
<tr><td colspan="2">Modifiers, name, super class, interfaces</td></tr>
<tr><td colspan="2">Constant pool: numeric, string and type constants</td></tr>
<tr><td colspan="2">Source file name (optional)</td></tr>
<tr><td colspan="2">Enclosing class reference</td></tr>
<tr><td colspan="2">Annotation*</td></tr>
<tr><td colspan="2">Inner class*</td></tr>
<tr><td rowspan="3">Field*</td><td>Modifiers, name, type</td></tr>
<tr><td>Annotation*</td></tr>
<tr><td>Attribute*</td></tr>
<tr><td rowspan="4">Method*</td><td>Modifiers, name, return and parameter types</td></tr>
<tr><td>Annotation*</td></tr>
<tr><td>Attribute*</td></tr>
<tr><td>Compiled code</td></tr>
</table>

- ตัวกำหนดสิทธิการเข้าถึง (Modifier) ชื่อ คลาสแม่ และอินเตอร์เฟส
- ตัวแปรที่ถูกประกาศ
- เมธอดหรือฟังก์ชั่นที่ถูกสร้าง

โดยคลาสที่ถูกแปลนั้นจะถูกเก็บอยู่ใน constant pool โดยจะประกอบไปด้วยข้อมูลที่เป็นตัวเลข สตริง ค่าคงที่ต่างๆ ที่อยู่ในคลาส โดยค่าคงที่ต่างๆ นั้นจะถูกตั้งค่าเพียงครั้งเดียวและจะถูกทำการอ้างอิงจากส่วนอื่นๆ ภายในคลาส โดยความแตกต่างของคลาสที่ถูกแปลกับยังไม่ได้ถูกแปลคือลักษณะของข้อมูลต่างๆ จะถูกแสดงแตกต่างกัน โดยคลาสที่ถูกแปลนั้นจะถูกแสดงในลักษณะข้อมูลที่เป็นรูปแบบภายใน อย่างเช่น ชื่อคลาสจะถูกแสดงโดยตรงแต่เครื่องหมายที่เป็นจุด (.) จะถูกแทนด้วยเครื่องหมาย / ดังนั้นลักษณะข้อมูลภายในของ คลาส String คือ java/lang/String เป็นต้น

โดยลักษณะของข้อมูลภายในที่เป็นแบบข้อมูลพื้นฐาน (Primitive data type) ชนิดต่างๆ และอาเรย์นั้นถูกแสดงอยู่ในตารางที่ 2.3

ตารางที่ 2.3 แสดงโครงสร้างการเปลี่ยนข้อมูลพื้นฐานของ Java

ชนิดของข้อมูล	รูปแบบภายใน
boolean	Z
char	C
byte	B
short	S
int	I
float	F
long	J
double	D
Object	Ljava/lang/Obect;
int[]	[I
Object [] []	[[Ljava/lang/Object;

โดยในส่วนของเมธอดหรือฟังก์ชั่นต่างๆ นั้นจะถูกระบุด้วยรูปแบบภายในที่แตกต่างกันออกไป โดยในตารางรูปที่ 2.4 นั้นแสดงตัวอย่างเบื้องต้นของรูปแบบในส่วนของเมธอด

ตารางที่ 2.4แสดงโครงสร้างการเปลี่ยนข้อมูลเมธอดของ Java

ลักษณะของเมธอด	รูปแบบภายใน
void m(int i, float f)	(IF)V
int m(Object o)	(Ljava/lang/Object;)I
int[] m(int i, String s)	(ILjava/lang/String;)[I
Object m(int[] i)	([I)Ljava/lang/Object;

2.7 งานวิจัยที่เกี่ยวข้อง

การขยายความสามารถของภาษาโดยอ้างอิงความสามารถจากภาษาหลักนั้นถูกนำมาเป็นวิธีการหลักซึ่งหลายๆ ภาษาได้นำวิธีการนี้มาเพื่อมาออกแบบและพัฒนาภาษาให้เหมาะสมกับงานที่แตกต่างกันดังนี้

2.7.1 XTC (eXTensible Compiler)

XTC [42] เป็นโครงการที่สำรวจการเขียนโปรแกรมแบบใหม่รวมไปถึงเครื่องมือเพื่อปรับปรุงวิธีการสร้าง ความปลอดภัย ประสิทธิภาพของระบบที่ซับซ้อน โดยวิธีการดังกล่าวถูกนำไปสร้างภาษาและตัวแปลภาษาอย่าง SuperC [43] ที่ขยายการทำงานของภาษา C เพื่องาน

ทางด้านเวลา หรือภาษา Fortress และภาษา Matchete โดยภาษา Fortress [44] นั้นเป็นภาษาที่ถูกออกแบบสำหรับงานทางด้าน work-stealing และการขยายคำนิยาม (extensible definition) ผ่านทางช่องการทำงานที่เตรียมไว้ สำหรับภาษา Matchete [45] นั้นเป็นภาษาที่ถูกออกแบบสำหรับงานทางด้านเปรียบเทียบรูปแบบ (Pattern matching)

2.7.2 JastAddJ

JastAddJ [29] นี้เป็นการออกแบบภาษา Java โดยใช้วิธีการขยายความสามารถของภาษา Java1.4 ก่อนและทำการขยายความสามารถของภาษา Java5 เพิ่มลงไป วิธีการขยายนี้จะจัดการแยกส่วนของภาษาหลักออกจากภาษาที่ถูกขยายออกจากกันและทำการสร้างตัวแปลภาษาด้วยเครื่องมืออย่าง JastAdd ซึ่งทำงานด้วยไวยากรณ์ข้อมูลแบบอ้างอิง (Reference Attribute Grammar, REGs) เพื่อกำหนดจุดอ้างอิงของการตรวจสอบความหมายของภาษา [46, 47]

2.7.3 Jaco

Jaco [48] เป็นการขยายตัวแปลภาษา Java เพื่อสนับสนุนการทำงานทางด้านข้อมูลทางพีชคณิต โดยชนิดของข้อมูลนี้จะถูกแปลงเป็นข้อมูลพื้นฐานและคลาสของภาษา Java ในภายหลัง

2.7.4 Fuji

Fuji [49] เป็นการขยายภาษา Java เพื่องานทางด้านลักษณะงาน (Feature-oriented programming) โดยมุ่งเน้นการจัดการงานออกแบบส่วนต่างๆ ซึ่งแต่ละส่วนนั้นจะเป็นระบบที่จัดการความต้องการ (Requirement) แต่ละส่วนของระบบโดยส่วนนี้สามารถเพิ่มหรือลดได้

งานเหล่านี้คือส่วนหนึ่งของงานวิจัยที่นำวิธีการขยายความสามารถของภาษาเพื่อตอบสนองต่อปัญหาที่เกิดขึ้น โดยวิธีการขยายความสามารถจะช่วยลดช่องว่างของวิธีการและเวลาสำหรับเรียนรู้และพัฒนาโปรแกรมของภาษาเหล่านั้น

บทที่ 3
วิธีดำเนินการ

ในบทนี้จะแสดงให้เห็นถึงปัญหาที่เกิดขึ้นเนื่องจากการขยายลักษณะภาษาที่มีความคล้ายกันระหว่างภาษา Java ที่ถูกใช้เป็นภาษาหลักและภาษา Groovy ที่ถูกนำเอาความสามารถของภาษามาทำการขยายลงเพื่อเพิ่มลักษณะการทำงานให้ภาษา Java และนำเสนอแนวทางการแก้ไขปัญหาที่เกิดจากลักษณะของภาษาที่คล้ายกันเพื่อรองรับรูปแบบการขยายภาษา

3.1 ปัญหาของการขยายภาษาที่คล้ายกัน

เพื่อให้การทำงานของภาษา Java สามารถทำงานกับภาษาเฉพาะทางได้อย่างเหมาะสม จึงมีความจำเป็นต้องมีการขยายความสามารถของภาษา โดยการขยายความสามารถของภาษาสำหรับการเขียนโปรแกรมนั้นสามารถทำได้ใน 2 ลักษณะดังที่กล่าวไปแล้ว เพื่อให้การขยายความสามารถของภาษาสามารถรองรับกับภาษาเฉพาะทางต่างๆ นั้นวิธีการที่เหมาะสมคือการขยายความสามารถในรูปแบบภายนอก โดยลักษณะความสามารถในการรองรับการทำงานของภาษาเฉพาะทางได้อย่างเหมาะสมและสามารถตอบสนองได้ในหลายๆ ภาษานั้นภาษา Groovy ได้นำเสนอแนวทางการทำงานที่น่าสนใจไว้

ภาษา Groovy นั้นเป็นภาษาสำหรับการเขียนโปรแกรมแบบพลวัตร [9, 50] (Dynamic programming language) ที่สามารถทำงานบน JVM ได้ ซึ่งภาษา Groovy นั้นใช้ภาษา Java เป็นรากฐานในการสร้างภาษาและได้ทำการขยายความสามารถของภาษาที่เหมือนกับภาษาอย่างภาษา Python, Ruby [51] และ Smalltalk [52] โดยการสนับสนุนการทำงานกับภาษาเฉพาะทางและไวยากรณ์ที่กระชับเพื่อให้การอ่านโค้ด และการบำรุงรักษาโค้ดทำได้ง่ายขึ้น นอกจากนี้ภาษา Groovy ยังสามารถรวมการทำงานได้กับคลาสของ Java และชุดคำสั่งต่าง อีกทั้งยังทำการแปลโค้ดให้เป็น Java ไบต์โค้ดได้โดยตรง

การที่ภาษา Groovy มีโครงสร้างที่สามารถตอบสนองต่อการทำงานด้านภาษาเฉพาะทางได้อย่างหลากหลาย และสามารถทำงานได้บนโครงสร้างของ JVM อีกทั้งยังสามารถสนับสนุนการทำงานในรูปแบบของภาษา Java ได้นั้นทำให้ภาษา Groovy เหมาะสมกับการนำไปพัฒนาโปรแกรม แต่เนื่องจากไวยากรณ์บางส่วนของภาษา Groovy นั้นถูกจัดการให้อยู่ในรูปแบบกะทัดรัดและกระชับต้องทำการตัดไวยากรณ์และไม่สนับสนุนการทำงานบางรูปแบบของภาษา Java

ปัญหาของภาษา Groovy นั้นเกิดจากรูปแบบที่มีความกะทัดรัดและกระชับเพราะทำให้ภาษา Groovy ไม่สามารถสนับสนุนการทำงานของภาษา Java ได้อย่างเต็มรูปแบบ เนื่องจากการที่จะทำให้ภาษามีความกระชับจากโครงสร้างของไวยากรณ์เดิมนั้นจะต้องทำการลดรูปของไวยากรณ์เดิม และเนื่องจากภาษามีความคล้ายกันเพราะทำการพัฒนามาจากรากของภาษาเดียวกันคือภาษา Java ถ้าไม่ทำการลดรูปแล้วจะทำให้ภาษาเกิดความกำกวมจนทำให้ไม่สามารถแยกแยะและดำเนินการสร้างภาษา Groovy ได้

3.1.1 ปัญหาของรูปแบบการจบประโยคของไวยากรณ์ (Problem of end of statement)

ปัญหาที่เกิดจากรูปแบบที่กะทัดรัดจากไวยากรณ์ของภาษา Groovy ที่ก่อให้เกิดลักษณะที่สร้างความสับสนให้กับไวยากรณ์ของภาษา Java คือรูปแบบการจบประโยคของไวยากรณ์ ซึ่งโดยทั่วไปนั้นภาษา Java จะใช้เครื่องหมายอัฒภาค (;) สำหรับเป็นตัวบอกจุดสิ้นสุดของประโยคดังตัวอย่าง

```
int sample_0 = 0 ;
```

แต่สำหรับภาษา Groovy นอกจากจะใช้เครื่องหมายอัฒภาคเป็นตัวบอกจุดสิ้นสุดของประโยคแล้วยังใช้การขึ้นบรรทัดใหม่ (Line terminator) สำหรับการบอกจุดสิ้นสุดของประโยคด้วยเช่นกันดังตัวอย่าง

```
int sample_1 = 0
```

รูปแบบลักษณะการจบประโยคดังข้างต้นของภาษา Groovy ทำให้เกิดความยุ่งยากขึ้นเมื่อนำมารวมกับไวยากรณ์ของภาษา Java แบบปกติเพราะอาจตีความประโยคคำสั่งผิดพลาด เช่น

```
int sample_2
= 0 ;
```

รูปประโยคดังข้างต้นถ้าเป็นไวยากรณ์ของภาษา Java จะตีความของประโยคคำสั่งว่าเป็นการประกาศตัวแปรแบบกำหนดค่า แต่ถ้าเป็นไวยากรณ์ของภาษา Groovy จะตีความหมายของประโยคคำสั่งดังกล่าวว่าเป็นการประกาศตัวแปร และเมื่อมีการขึ้นบรรทัดใหม่ก็ถือว่าประโยคคำสั่งข้างต้นนั้นเสร็จสมบูรณ์แล้ว ดังนั้นค่าที่ตามมานั้นถือว่าผิดไวยากรณ์ของภาษา ซึ่งความจริงประโยคคำสั่งดังกล่าวนั้นเป็นประโยคเดียวกัน

ปัญหาที่กล่าวมาข้างต้นนั้นเกิดจากการที่มีการกำหนดรูปประโยคที่กระชับกะทัดรัดทำให้การตีความหมายของไวยากรณ์ดังกล่าวมีความผิดพลาดดังกล่าวขึ้น และไม่สามารถกำหนดและตรวจสอบในส่วนของการกำหนดโครงร่างของไวยากรณ์ภาษาได้

3.1.2 ปัญหาของรูปแบบการเรียกเมธอดแบบไม่มีวงลบ (Problem of method invocation without parenthesis)

โครงสร้างของภาษา Java แบบปกตินั้นจำเป็นต้องมีการเปิด-ปิดวงเล็บเพื่อกำหนดรูปแบบสำหรับการส่งค่าและเรียกใช้สำหรับเมธอดต่างๆ ดังนี้

```
void call(String s){
    …
}

String sample = "HelloWorld"
```

```
Call (sample) ;
```

เมื่อเทียบกับภาษา Groovy แล้วการส่งค่าและเรียกใช้สำหรับเมธอดต่างๆ นั้นสามารถกระทำได้ด้วยอีกวิธีการหนึ่งดังตัวอย่าง

```
void Call(String s){
    …
}

String sample = "HelloWorld"
Call sample ;
```

ด้วยไวยากรณ์ดังกล่าวอาจตีความได้ว่าเป็นการประกาศตัวแปรของคลาส Call ได้ในภาษา Java แบบปกติดัง

3.1.3 ปัญหาของการสร้างอาร์เรย์แบบคงที่ (Problem of static array initialization)

ปัญหาดังกล่าวเกิดขึ้นจากที่ภาษา Groovy มีรูปแบบให้มีการสร้างอาร์เรย์แบบคงที่แตกต่างจากภาษา Java กล่าวคือรูปแบบการสร้างอาร์เรย์แบบคงที่ของภาษา Java มีลักษณะดังตัวอย่างต่อไปนี้

```
int[] i = {2,3};
```

ในขณะที่รูปแบบการสร้างอาร์เรย์แบบคงที่ของภาษา Groovy มีลักษณะที่แตกต่างกันเล็กน้อยดังนี้

```
int[] i = [2,3];
```

เนื่องจากรูปแบบของการเปิด-ปิดปีกกานั้นถูกเตรียมไว้ใช้สำหรับการทำงานของส่วนการทำงานแบบปิด (Closure) โดยเฉพาะ ดังนั้นถ้ามีการใช้ปีกกาแบบภาษา Java จะทำให้เกิดความยุ่งยากในการแยกแยะความแตกต่างของไวยากรณ์มากขึ้น

3.1.4 ปัญหาของลำดับชั้นของโครงสร้างคลาส (Problem of the hierarchy of class structure)

เนื่องจากโครงสร้างที่กะทัดรัด โครงสร้างแบบปิด และโครงสร้างคลาสแบบภายใน (Inner class) นั้นสามารถถูกกำหนดได้แทบทุกส่วนของโครงสร้างคลาส ลำดับชั้นของโครงสร้างนั้นจำเป็นจะต้องถูกระบุให้มีความชัดเจนถ้าไม่เช่นนั้นแล้วตัวตัดคำจะไม่สามารถจำแนกแยกแยะคำสำคัญต่างๆ ได้อย่างถูกต้อง และในเมื่อตัวตัดคำไม่สามารถจำแนกแยกแยะได้แล้วนั้น โครงสร้างไวยากรณ์จะไม่สามารถถูกกำหนดได้ เพราะภาษามีความใกล้เคียงกันมากจนทำให้เกิดความสับสน ดังนั้นการกำหนดลำดับชั้นของโครงสร้างนี้ถือเป็นปัญหาสำคัญปัญหาหนึ่งในการออกแบบภาษาที่มีความใกล้เคียงกัน

3.2 ภาษา JGroovy

ภาษา JGroovy นั้นคือนำภาษา Java ทั้งหมดมาขยายความสามารถและโครงสร้าง โดยภาษา Java นั้นถูกใช้เพื่อเป็นรากของภาษา JGroovy และนำโครงสร้างและความสามารถของภาษา Groovy มาขยายเพิ่มลงไป แต่การขยายภาษา Groovy ลงไปนั้นได้ทำให้โครงสร้างของภาษาบางส่วนของภาษา Java เกิดปัญหาขึ้นดังที่ได้กล่าวมาแล้วในข้างต้น นอกจากนี้ภาษา Groovy เองนั้นมีโครงสร้างของภาษาที่สนับสนุนการทำงานแบบเมตา (Meta Object Protocol, MOP) โดยการออกแบบให้มีอินเตอร์เฟสเพื่อรองรับ API สำหรับสนับสนุนการใช้งานโดยลูกข่ายอื่นๆ สำหรับเรียกใช้ MOP [53]

เนื่องจากว่าภาษา Groovy นั้นมีโครงสร้างของภาษาที่สนับสนุนการทำงานของภาษา Java อยู่แล้วจึงไม่จำเป็นต้องนำโครงสร้างของภาษา Groovy ทั้งหมดมาขยายความสามารถลงไปในภาษา Java การนำโครงสร้างของภาษา Groovy มาขยายความสามารถลงไปในภาษา Java เพื่อเป็นภาษา JGroovy นั้นได้ใช้โครงสร้างบางส่วนของภาษา Groovy ที่ไม่ปรากฏในภาษา Java ปกติ โดยโครงสร้างของภาษา Groovy ที่นำมาขยายนั้นมีลักษณะดังต่อไปนี้

- โครงสร้างแบบปิด (Closure)
- โครงสร้างแบบธรรมชาติของ List และ Maps (Native syntax for lists and maps)
- โครงสร้างแบบธรรมชาติที่สนับสนุนการทำงานของนิพจน์ปกติ (Regular expression)
- โครงสร้างที่ฝังการทำงานของสตริง
- โครงสร้างการทำงานแบบทางเลือกหลายทางและโครงสร้างพ้องรูปของการวนซ้ำ (Switch statement and polymorphic iteration)
- โครงสร้างที่กระชับสำหรับคลาสของ Java (Smart syntax for Java bean)
- โครงสร้างที่เหมาะสมปลอดภัยสำหรับการจัดการปัญหา

ดังนั้นเพื่อสนับสนุนการขยายภาษา Groovy เพิ่มลงไปในภาษา Java จำเป็นต้องปรับปรุงโครงสร้างของตัวแปลภาษาในส่วนต่างๆ ดังต่อไปนี้

- ที่ส่วนการวิเคราะห์คำต้องทำการเพิ่มคำเฉพาะ เช่น คำเฉพาะแบบพลวัตร (Dynamic type) โครงสร้างที่เหมาะสมปลอดภัยสำหรับการจัดการปัญหา (Safe navigation) และโครงสร้างแบบธรรมชาติที่สนับสนุนการทำงานของนิพจน์ปกติ เป็นต้น
- ในส่วนของการตรวจสอบไวยากรณ์ต้องทำการเพิ่มโครงสร้างไวยากรณ์ของภาษา Groovy ที่ไม่ปรากฏในภาษาหลัก
- สุดท้ายในส่วนของการตรวจสอบความหมายของภาษานั้นจะต้องขยายความหมายภาษา Groovy ที่ไม่ปรากฏในภาษาหลักลงไปเพิ่มอีกเช่นเดียวกัน

3.3 การออกแบบตัวแปลภาษา JGroovy เพื่อรองรับการขยายภาษาที่คล้ายกัน

โดยทั่วไปนั้นไวยากรณ์ของภาษาจะเป็นโครงร่างที่สำคัญสำหรับการคัดแยกความแตกต่างระหว่างคำสั่ง หรือรูปประโยคที่ใช้กำหนดการทำงานแต่ละชนิด การที่ไวยากรณ์ของภาษาสามารถคัดแยกได้นั้นเป็นเพราะว่าโครงสร้างของคำสั่งต่างๆ เหล่านั้นมีความแตกต่างกันอย่างชัดเจนสำหรับการทำงานในแต่ละชนิด โครงสร้างของภาษาที่แตกต่างกันอย่างชัดเจน และสามารถแยกความแตกต่างได้

ในระดับของไวยากรณ์นั้นทำให้ภาระงานสำหรับการในการแยกแยะโครงสร้างของภาษาตกอยู่ที่ส่วนของการตรวจสอบไวยากรณ์เพียงส่วนเดียวเท่านั้น

ดังนั้นเมื่อมีออกแบบเพื่อสร้างหรือขยายภาษาสำหรับการเขียนโปรแกรมคอมพิวเตอร์ใหม่ขึ้นมา ซึ่งโครงสร้างของภาษาใหม่นั้นมีความแตกต่างเล็กน้อยซึ่งพบเพียงบางส่วนของภาษา พบว่าส่วนของการตรวจสอบไวยากรณ์นั้นไม่อาจทำการคัดแยกความแตกต่างของภาษาได้

เพื่อช่วยเหลือการทำงานของส่วนการตรวจสอบไวยากรณ์ ส่วนการคัดแยกคำจำเป็นต้องได้รับการออกแบบให้จำแนกรูปแบบของคำเพื่อลดความซับซ้อน ซึ่งปกติโดยทั่วๆ ไปแล้วส่วนการคัดแยกคำมีหน้าที่คัดแยกคำที่ได้กำหนดไว้สำหรับภาษาแต่ละชนิดเท่านั้น ดังนั้นการช่วยเหลือส่วนการตรวจสอบไวยากรณ์แล้วส่วนการคัดแยกคำต้องทำงานได้อย่างเหมาะสม เพื่อทำการคัดแยกคำที่ได้กำหนดไว้และกำหนดรูปแบบของไวยากรณ์ที่ใกล้เคียงกันให้ได้ในเวลาเดียวกัน โดยรูปแบบการทำงานของตัวแปลภาษาส่วนหน้า (Front end) จะแบ่งการทำงานออกเป็น 3 ส่วนเพื่อรองรับการขยายภาษาที่คล้ายกัน

3.3.1 ตัวคัดแยกคำสถานะจำกัดแบบลำดับชั้น (Stack Machine Analyzer)

เพื่อช่วยสนับสนุนต่อการคัดแยกโครงสร้างของไวยากรณ์ที่มีความคล้ายคลึงกัน ส่วนของตัวคัดแยกคำนั้นจำเป็นต้องได้รับการกำหนดทิศทางของแต่ละสถานะให้เหมาะสม เพื่อรองรับความหมายของภาษาหลักคือภาษา Java และภาษาที่นำมาขยายความสามารถเพิ่มลงไปในภาษาหลักอย่างภาษา Groovy ซึ่งจะต้องใช้รูปแบบของการทำงานของสถานะจำกัด [54] เข้ามาร่วมโดยกำหนดรูปแบบออกเป็น 6 แถวข้อมูล (Tuple) (S, S0, Σ, Λ, T, G) เพื่อทำงานร่วมกันกับตัวจัดลำดับชั้น (Stack) โดยข้อมูลของทั้ง 6 แถวข้อมูลเป็นดังต่อไปนี้ [55]

- สัญลักษณ์ S คือ เซตของสถานะ
- สัญลักษณ์ S0 คือ สถานะเริ่มต้นซึ่งเป็นหนึ่งในสมาชิกของเซตทั้งหมด
- สัญลักษณ์ Σ คือ สถานะจำกัดซึ่งคอยรับข้อมูล
- สัญลักษณ์ Λ คือสถานะจำกัดซึ่งคอยส่งข้อมูล
- สัญลักษณ์ T คือฟังก์ชั่นการเปลี่ยนสถานะซึ่งคอยกำหนดค่าเข้ามาว่าจะไปทำงานในสถานะใด
- สัญลักษณ์ G คือฟังก์ชั่นที่คอยกำหนดสถานะว่าเมื่อรับค่าเข้ามาแล้วค่าที่ออกจากฟังก์ชั่นนั้นๆ จะเป็นค่าใดและไปทำงานที่สถานะใดต่อไป

โดยได้ทำการกำหนดการทำงานออกเป็นสถานะทั้งหมด 55 สถานะและตัวกำหนดลำดับชั้นเพื่อทำหน้าที่ต่างๆ ดังต่อไปนี้

3.1.1.1 ตัวจัดลำดับชั้น (Stack)

การที่โครงสร้างภาษาที่กะทัดรัด โครงสร้างแบบปิด และโครงสร้างคลาสแบบภายในจากภาษา Java และ Groovy ซึ่งเมื่อนำมาขยายความสามารถเป็นภาษา JGroovy นั้นทำให้เกิดปัญหาดังที่ได้กล่าวไว้ในข้างต้น ดังนั้นการสร้างตัวจัดลำดับชั้นนั้นเพื่อทำให้ตัวคัดแยกคำสามารถทำงานได้อย่างถูกต้องโดยการจัดการลำดับของคำ

(Token) ในขณะนั้นว่าอยู่ในสถานะใด เพื่อกำหนดว่าต่อไปคำใหม่ที่เข้ามานั้นจะถูกกำหนดให้ไปอยู่ในสถานะที่ถูกต้องและเหมาะสม

ตัวจัดลำดับชั้นจะเก็บค่าที่เป็นตัวเลขเพื่อบอกระดับของความลึกที่ข้อมูลนั้นๆ ดำเนินการอยู่ โดยเมื่อทำงานเสร็จสิ้นในระดับนั้นๆ แล้วจำทำการลบลดความลึกลง หรือเพิ่มความลึกอีกเมื่อข้อมูลนั้นๆ มีรูปแบบการสร้างคลาสภายในขึ้น

3.1.1.2 สถานะเริ่มต้น (Initial state)

สถานะนี้เป็นสถานะเริ่มต้นของการทำงานของตัวคัดแยกคำซึ่งลักษณะการทำงานของสถานะนี้จะคอยรับค่าใดๆ และกำหนดถึงสถานะเมื่อได้รับค่านั้นๆ แล้วดังตารางที่ 3.1

ตารางที่ 3.1 สัญลักษณ์ที่ถูกใช้ในสถานะเริ่มต้น

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“package”	คำสำคัญในการระบุ package	ไปสถานะ PACKAGE
“import”	คำสำคัญในการเรียกใช้งานคลาส	ไปสถานะ IMPORT
“;”	สิ้นสุดการทำงานของประโยค	คงอยู่ที่สถานะเดิม
.	ไม่มี	ไปสถานะ CLASS_INTERFACE_DECL

หมายเหตุ เครื่องหมายจุด (.) หมายถึงค่าใดๆ ที่ไม่ได้ถูกกำหนดขึ้นในสถานะนั้นๆ

เมื่อค่าที่รับเข้ามาเข้าเงื่อนไขสำหรับการทำงานของการคัดแยกคำแล้ว ค่าต่อไปจะถูกป้อนเข้ามาเรื่อยๆ เพื่อทำงานจนอักขระตัวสุดท้าย

3.1.1.3 สถานะ PACKAGE (Package state)

เมื่อได้ค่าจากสถานะเริ่มต้นแล้วในสถานะ PACKAGE นี้จะคอยคัดแยกลักษณะคำที่เกี่ยวข้องกับ PACKAGE โดยเฉพาะดังตารางที่ 3.2

ตารางที่ 3.2 สัญลักษณ์ที่ถูกใช้ในสถานะ PACKAGE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"\n|\r|\r\n"	สิ้นสุดการทำงานของประโยค	กลับไปที่สถานะเริ่มต้น
";"	สิ้นสุดการทำงานของประโยค	กลับไปที่สถานะเริ่มต้น
"."	เครื่องหมายจุด	คงอยู่ที่สถานะเดิม
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม

การทำงานของสถานะ PACKAGE นั้นตรงไปตรงมา ไม่มีความซับซ้อน ถึงแม้ว่าจะมีสัญลักษณ์สิ้นสุดการทำงานของประโยคทั้ง 2 ชนิดมาทำงานร่วมกันก็ตาม

3.1.1.4 สถานะ IMPORT (Import state)

การทำงานในสถานะนี้จะเหมือนกันกับสถานะ PACKAGE คือเมื่อได้รับค่าที่กำหนดมาจากสถานะเริ่มต้นแล้ว สถานะ IMPORT นี้จะคอยคัดแยกลักษณะคำที่เกี่ยวข้องกับ IMPORT โดยเฉพาะดังตารางที่ 3.3

ตารางที่ 3.3 สัญลักษณ์ที่ถูกใช้ในสถานะ IMPORT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"\n|\r|\r\n"	สิ้นสุดการทำงานของประโยค	กลับไปที่สถานะเริ่มต้น
";"	เครื่องหมายอัฒภาค	กลับไปที่สถานะเริ่มต้น
"."	เครื่องหมายจุด	คงอยู่ที่สถานะเดิม
"*"	เครื่องหมายการคูณ	คงอยู่ที่สถานะเดิม
"static"	คำสำคัญ	คงอยู่ที่สถานะเดิม
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม

3.1.1.5 สถานะ CLASS_INTERFACE_DECL (Class and interface declaration state)

โดยทั่วไปนั้นภาษา Java อนุญาตให้มีการตั้งชื่อแพ็คเก็จและเรียกใช้คลาสจากที่อื่นได้โดยการใช้คำเฉพาะคือ `package` และ `import` แต่ถ้าไม่มีการระบุการทำงานดังกล่าวก็ไม่จำเป็นจะต้องเรียกใช้งานในคำเฉพาะนั้นๆ ดังนั้นสถานะนี้จึงไม่ได้รับค่าต่อมาจากสถานะ PACKAGE และ IMPORT แต่รับค่ามาจากสถานะเริ่มต้นเช่นเดียวกับ 2 สถานะแรกโดยสัญลักษณ์ที่ใช้ในสถานะ CLASS_INTERFACE_DECL นี้ถูกระบุอยู่ในตารางที่ 3.4

ตารางที่ 3.4 สัญลักษณ์ที่ถูกใช้ในสถานะ CLASS_INTERFACE_DECL

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“def”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“enum”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“abstract”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“class”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“final”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“private”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“protected”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“public”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“interface”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“strictfp”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“;”	เครื่องหมายอัฒภาค	กลับไปที่สถานะเริ่มต้น
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ไปสถานะ EXTEND_IMPLEMENTS_DECL
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	ไปสถานะ EXTEND_IMPLEMENTS_DECL
“.”	เครื่องหมายจุด	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น
“}”		ถ้าลำดับชั้นมากกว่า 1 ชั้นให้ดึงค่าจากลำดับชั้นบนสุดเพื่อไปสถานะดังกล่าว จากนั้นลบสถานะบนสุดออก

การทำงานของสถานะ CLASS_INTERFACE_DECL นั้นเริ่มมีความซับซ้อนขึ้นเนื่องจากมีการทำงานร่วมกับตัวจัดลำดับชั้นในส่วนของเครื่องหมายจุดและปีกกา

ปิดเป็นเพราะว่าลักษณะของสถานะนี้เป็นการสร้างส่วนหัวของคลาสดังตัวอย่างข้างล่าง

```
Public class Sample {
 …
}
```

ซึ่งสถานะ CLASS_INTERFACE_DECL นั้นสามารถถูกใช้ซ้ำเมื่อมีการสร้างคลาสแบบภายในได้ ซึ่งตัวจัดลำดับชั้นจะช่วยบอกระดับของการสร้างคลาสว่ามีการสร้างคลาสภายในหรือไม่และจัดการคืนสถานะที่เหมาะสมกลับไปโดยนำค่าจากตัวจัดลำดับ

3.1.1.6 สถานะ EXTEND_IMPLEMENTS_DECL (Extend and Implement declaration state)

เมื่อมีการกำหนดชื่อคลาสจากสถานะก่อนหน้าคือ CLASS_INTERFACE_DECL ทำให้สถานะนี้ต้องคอยตรวจสอบว่ามีการประกาศและใช้คำเฉพาะสำหรับการสืบทอดคลาสหรือขยายความสามารถคลาสด้วยอินเตอร์เฟสหรือไม่ โดยสถานะนี้จะคอยคัดแยกลักษณะคำที่เกี่ยวข้องกับ EXTEND_IMPLEMENTS_DECL โดยเฉพาะดังตารางที่ 3.5

ตารางที่ 3.5 สัญลักษณ์ที่ถูกใช้ในสถานะ EXTEND_IMPLEMENTS_DECL

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“extends”	คำสำคัญ	ไปสถานะ CLASS_INTERFACE_DECL
“implements”	คำสำคัญ	ไปสถานะ CLASS_INTERFACE_DECL
“,”	เครื่องหมายจุลภาค	ไปสถานะ CLASS_INTERFACE_DECL
“.”	เครื่องหมายจุด	ไปสถานะ GENERIC_DECL
“<”	เครื่องหมายน้อยกว่า	ไปสถานะ GENERIC_DECL และทำการใส่ค่าลำดับขั้นของสถานะ CLASS_INTERFACE_DECL ลงไปในตัวจัดลำดับ
“{”	เครื่องหมายปีกกาเปิด	ไปสถานะ CLASS_INTERFACE_BODY และทำการใส่ค่าลำดับขั้นของสถานะ CLASS_INTERFACE_DECL ลงไปในตัวจัดลำดับเมื่อพบว่าขนาดของตัวเก็บเป็นศูนย์

สถานะนี้มีทางเลือกของสถานะอื่นเพิ่มขึ้นอีก 2 สถานะคือสถานะ GENERIC_DECL และสถานะ CLASS_INTERFACE_BODY ซึ่งมีการจัดเก็บสถานะ CLASS_INTERFACE_DECL ลงไปในตัวจัดการลำดับขึ้นเมื่อมีการเปลี่ยนสถานะ

3.1.1.7 สถานะ GENERIC_DECL (Generic declaration state)

สถานะ GENERIC_DECL ใช้สำหรับคัดแยกคำของลักษณะการทำงานที่เป็น Generic ที่เป็นลักษณะการทำงานของ Java5 โดยทำการคัดแยกลักษณะคำที่เกี่ยวข้องดังตารางที่ 3.6

ตารางที่ 3.6 สัญลักษณ์ที่ถูกใช้ในสถานะ GENERIC_DECL

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“extends”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“?”	เครื่องหมายคำถาม	คงอยู่ที่สถานะเดิม
“,”	เครื่องหมายจุลภาค	คงอยู่ที่สถานะเดิม
“.”	เครื่องหมายจุด	คงอยู่ที่สถานะเดิม
“>”	เครื่องหมายมากกว่า	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก
.	ไม่มี	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก

3.1.1.8 CLASS_INTERFACE_BODY (Class and Interface body)

สถานะ CLASS_INTERFACE_BODY ถือว่าเป็นหนึ่งในสถานะหลักที่เกี่ยวข้องกับสถานะอื่นเป็นจำนวนมาก หน้าที่หลักของสถานะนี้จะคอยรองรับรูปแบบที่เกิดขึ้นภายในคลาสของภาษา JGroovy โดยทำการคัดแยกลักษณะคำที่เกี่ยวข้องดังตารางที่ 3.7

ตารางที่ 3.7 สัญลักษณ์ที่ถูกใช้ในสถานะ CLASS_INTERFACE_BODY

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“class”	คำสำคัญ	ไปสถานะ CLASS_INTERFACE_DECL และทำการจัดเก็บลำดับของ CLASS_INTERFACE_BODY ลงไปในตัวจัดระดับ
“interface”	คำสำคัญ	ไปสถานะ CLASS_INTERFACE_DECL และทำการจัดเก็บลำดับของ CLASS_INTERFACE_BODY ลงไปในตัวจัดระดับ
“enum”	คำสำคัญ	ไปสถานะ ENUM และทำการจัดเก็บลำดับของ CLASS_INTERFACE_BODY ลงไปในตัวจัดระดับ
“void”, “long”, “boolean”, “double”, “short”, “byte”, “float”, “int”, “char”, “def”	คำสำคัญ	ไปสถานะ VARIABLE_METHOD_NAME และทำการจัดเก็บลำดับของ CLASS_STATE ลงไปในตัวจัดระดับ
“switch”	คำสำคัญ	ไปสถานะ BEFORE_CONDITION และทำการจัดเก็บลำดับของ CLASS_INTERFACE_BODY ลงไปในตัวจัดระดับ

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“abstract”, “static”, “private”, “protected”, “public”, “native”, “transient”, “strictfp”, “final”, “volatile”, “synchronized”	คำสำคัญ	คงอยู่ที่สถานะเดิม
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ไปสถานะ VARIABLE_METHOD_NAME และทำการจัดเก็บลำดับของ CLASS_STATE ลงไปในตัวจัดระดับ
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	ไปสถานะ VARIABLE_METHOD_NAME และทำการจัดเก็บลำดับของ CLASS_STATE ลงไปในตัวจัดระดับ
“[”, “]”	เครื่องหมายวงเล็บ	คงอยู่ที่สถานะเดิม
“{”	เครื่องหมายปีกกาเปิด	ไปสถานะ STATIC_METHOD_BODY และทำการจัดเก็บลำดับของ CLASS_INTERFACE_BODY ลงไปในตัวจัดระดับ
“}”	เครื่องหมายปีกกาปิด	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก แต่ถ้าขนาดของตัวจัดลำดับเป็นศูนย์ให้กลับไปที่สถานะ CLASS_INTERFACE_DECL
“;”	เครื่องหมายอัฒภาค	คงอยู่ที่สถานะเดิม

จะเห็นได้ว่ามีสถานะ CLASS_STETE เข้ามาเกี่ยวข้อง โดยสถานะดังกล่าวถูกออกแบบมาเพื่อจัดการปัญหาการจบประโยคของในส่วนของคลาส นอกจากนี้สถานะอย่างเช่น VARIABLE_METHOD_NAME STATIC_METHOD_BODY และ BEFORE_CONDITION ถูกกำหนดมาเพื่อการประกาศตัวแปรระดับสูง (Global variable)โครงสร้างภายในเมธอดและเงื่อนไขตามลำดับ

3.1.1.9 สถานะ ENUM (Enum state)

enum เป็นคำสำคัญที่เพิ่มขึ้นมาสำหรับการทำงานในส่วนของภาษา Java5 โดยสถานะนี้จะทำการคัดแยกลักษณะคำที่เกี่ยวข้องดังตารางที่ 3.8

ตารางที่ 3.8 สัญลักษณ์ที่ถูกใช้ในสถานะ ENUM

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ไปสถานะ ENUM_BODY และทำการจัดเก็บลำดับของ CLASS_INTERFACE_DECL ลงไปในตัวจัดระดับ
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม

3.1.1.10 สถานะ ENUM_BODY (Enum body state)

เพื่อรองรับการทำงานของ enum ซึ่งสามารถสร้างโครงสร้างจัดการข้อมูลที่เป็นลักษณะที่สามารถกำหนดรูปแบบเองได้ดังนั้นสถานะ ENUM_BODY จำเป็นต้องจัดการลักษณะคำที่เกี่ยวข้องดังตารางที่ 3.9

ตารางที่ 3.9 สัญลักษณ์ที่ถูกใช้ในสถานะ ENUM_BODY

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ทำการจัดเก็บลำดับของ ENUM_BODY ลงไปในตัวจัดระดับ

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT และทำการจัดเก็บลำดับของ ENUM_BODY ลงไปในตัวจัดระดับ
“,”	เครื่องหมายจุลภาค	คงอยู่ที่สถานะเดิม
“;”	เครื่องหมายอัฒภาค	ลบลำดับชั้นในตัวจัดเก็บออก 1 ชั้น และไปสถานะ CLASS_INTERFACE_BODY
“}”	เครื่องหมายปีกกาปิด	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม

3.1.1.11 สถานะ ARGUMENT (Argument state)

สถานะ AGUMENT มีการระบุคำเพื่อรองรับการทำงานในส่วนการรับค่าของเมธอด รวมถึงใช้ในการระบุในส่วนของ constructor ด้วยเช่นกัน โดยมีการกำหนดรูปแบบที่เกี่ยวข้องดังตารางที่ 3.10

ตารางที่ 3.10 สัญลักษณ์ที่ถูกใช้ในสถานะ ARGUMENT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	มีการเปลี่ยนสถานะหลายเงื่อนไข โดยขึ้นอยู่กับค่าที่ถูกเก็บอยู่ในตัวจัดลำดับชั้นซึ่งมีเงื่อนไขเช่น ถ้าลำดับชั้นในตัวจัดเก็บอยู่ที่สถานะ INIT ให้ไปที่สถานะ CLASS_INTERFACE_BODY เป็นต้น

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“ “def”,final”, “long”, “boolean”, “double”, “short”, “byte”, “float”, “int”, “char”,	คำสำคัญ	คงอยู่ที่สถานะเดิม
“[”	เครื่องหมายวงเล็บเปิด	ถ้าค่าเป็นสุดที่อยู่ในตัวกำหนดลำดับเป็นสถานะ INIT ให้ลบทิ้งและใส่ค่า ARGUMENT ลงไปแทน จากนั้นไปที่สถานะ ARRAY_INIT
“]”	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิม
“<”	เครื่องหมายน้อยกว่า	ไปสถานะ GENERIC_DECL และทำการใส่ค่าลำดับขั้นของสถานะ ARGUMENT ลงไปในตัวจัดลำดับ
“(”	เครื่องหมายวงเล็บเปิด	ถ้าค่าเป็นสุดที่อยู่ในตัวกำหนดลำดับเป็นสถานะ INIT ให้ลบทิ้งและใส่ค่า ARGUMENT และ INIT ลงไปตามลำดับ
“)”	เครื่องหมายวงเล็บปิด	มีการเปลี่ยนสถานะหลายเงื่อนไขโดยขึ้นอยู่กับค่าที่ถูกเก็บอยู่ในตัวจัดลำดับชั้นซึ่งมีเงื่อนไข
“}”	เครื่องหมายปีกกาปิด	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก
.	ไม่มี	ไปสถานะ LITERAL และใส่ค่าลำดับขั้นของสถานะ ARGUMENT ลงไปในตัวจัดลำดับ

สถานะนี้เป็นหนึ่งในสถานะที่สำคัญเนื่องจากทางเลือกของการกำหนดเส้นทางของสถานะต่อไปขึ้นอยู่กับ 2 ปัจจัยหลักคือข้อมูลนำเข้าและลำดับชั้นที่อยู่ใน

ตัวจัดลำดับชั้น ดังนั้นทางเลือกของการกำหนดเส้นทางของสถานะต่อไปต้องพิจารณาปัจจัยทั้ง 2 ชนิดพร้อมกัน

3.1.1.12 สถานะ VARIABLE_METHOD_NAME (Variable and method name state)

สถานะ VARIABLE_METHOD_NAME เป็นอีกหนึ่งสถานะที่ต่อเนื่องที่ถูกกำหนดให้มาสนับสนุนการแก้ปัญหาของการจบประโยคของไวยากรณ์และปัญหาการสร้างอาร์เรย์แบบคงที่ เพราะสถานะนี้จะคอยทำหน้าที่ระบุชื่อของคลาสและตัวแปรทุกๆ ตัวภายในภาษา JGroovy ซึ่งหลังจากการระบุชนิดของชื่อแล้วจะต้องคอยตรวจสอบปัญหาของการจบประโยคของไวยากรณ์ โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.11

ตารางที่ 3.11 สัญลักษณ์ที่ถูกใช้ในสถานะ VARIABLE_METHOD_NAME

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT และทำการจัดเก็บลำดับของ BEFORE_BODY ลงไปในตัวจัดระดับ
“.”	เครื่องหมายจุด	ทำการลบค่าในตัวจัดระดับจนถึงระดับของสถานะ CLASS_STATE และจากนั้นไปสถานะ CLASS_INTERFACE_BODY
“<”	เครื่องหมายน้อยกว่า	ไปสถานะ GENERIC_DECL และทำการใส่ค่าลำดับขั้นของสถานะ VARIABLE_METHOD_NAME ลงไปในตัวจัดลำดับ
“[”	เครื่องหมายวงเล็บเปิด	คงอยู่ที่สถานะเดิม
“]”	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิมและใส่ค่าลำดับขั้นของสถานะ ARRAY_INIT ลงไปในตัวจัดลำดับ
“new”	คำสำคัญ	ไปสถานะ INIT และทำการใส่ค่าลำดับขั้นของสถานะ SEPARATOR และ INIT ลงไปในตัวจัดลำดับ
คำสำคัญที่เหลือทั้งหมด	คำสำคัญ	ไปที่สถานะ SEPARATOR

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ไปที่สถานะ SEPARATOR
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	ไปที่สถานะ SEPARATOR

หมายเหตุ คำสำคัญที่เหลือทั้งหมดสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.13 สถานะ SEPARATOR (Separator state)

สถานะนี้เป็นสถานะกลางที่ใช้ในการแก้ปัญหาที่เกิดจากการจบประโยคของไวยากรณ์ที่คล้ายกัน โดยต้องทำงานร่วมกันกับหลายๆ สถานะก่อนหน้า และมาทำการหาจุดจบของรูปประโยคแบบไม่ใช้เครื่องหมายที่สถานะนี้โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.12

ตารางที่ 3.12 สัญลักษณ์ที่ถูกใช้ในสถานะ VARIABLE_METHOD_NAME

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"\n|\r|\r\n"	สิ้นสุดการทำงานของประโยค	ตรวจสอบค่าในตัวจัดลำดับขั้นจนกว่าจะพบส่วนที่เป็นลำดับของ CLASS_STATE METHOD_STATE หรือ CLOSURE_STATE เมื่อพบสถานะดังกล่าวข้างต้นสถานะใดสถานะหนึ่งให้ไปสถานะนั้น
","	เครื่องหมายจุลภาค	ไปที่สถานะ VARIABLE_METHOD_NAME
"..", "..<"	เครื่องหมายจุดสำหรับการทำงานของภาษา Groovy ที่เพิ่มเข้ามา	ไปที่สถานะ LITERAL และใส่ค่าลำดับขั้นของสถานะ SEPARATOR ลงไปในตัวจัดลำดับ
"."	เครื่องหมายจุด	ไปที่สถานะ VARIABLE_METHOD_NAME แบบมีเงื่อนไข
"["	เครื่องหมายวงเล็บเปิด	คงอยู่ที่สถานะเดิม
"]"	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิมและใส่ค่าลำดับขั้นของสถานะ ARRAY_INIT ลงไปในตัวจัดลำดับ

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	มีการเปลี่ยนสถานะหลายเงื่อนไข โดยขึ้นอยู่กับค่าที่ถูกเก็บอยู่ในตัวจัดลำดับชั้นซึ่งมีเงื่อนไขเช่น ถ้าลำดับชั้นในตัวจัดเก็บอยู่ที่สถานะ INIT ให้ไปที่สถานะ CLASS_INTERFACE_BODY เป็นต้น
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT และทำการจัดเก็บลำดับของ SEPARATOR และBEFORE_BODY ลงไปในตัวจัดระดับ
“)”	เครื่องหมายวงเล็บปิด	ไปที่สถานะบนสุดของตัวจัดลำดับ
“;”	เครื่องหมายอัฒภาค	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก
“}”	เครื่องหมายปีกกาปิด	ไปสถานะที่อยู่บนสุดของตัวจัดลำดับชั้น จากนั้นลบสถานะบนสุดออก
.	ไม่มี	ไปสถานะ OPERATOR

สถานะ SEPARATOR เป็นสถานะที่เป็นตัวเชื่อมต่อกับหลายๆ สถานะโดยเงื่อนไขของการเปลี่ยนสถานะนั้นขึ้นอยู่กับค่าในตัวจัดลำดับเป็นสำคัญ เพราะจะเป็นตัวบ่งชี้ว่าการคัดแยกคำในปัจจุบันถูกทำอยู่ในส่วนไหนของข้อมูล

3.1.1.14 สถานะ OPERATOR (Operator state)

สถานะ OPERATOR เป็นสถานะสุดท้ายเพื่อกำหนดรูปแบบการปิดประโยคของภาษา JGroovy ซึ่งถ้าไม่พบตัวดำเนินการนั้นหมายความว่ามีรูปแบบการปิดประโยคแบบไม่ใช้เครื่องเกิดขึ้น ซึ่งจะมีการกำหนดว่าเป็นลักษณะดังกล่าวในสถานะต่อไปอย่าง CLASS_STATE METHOD_STATE และ CLOSURE_STATE โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.13

ตารางที่ 3.13 สัญลักษณ์ที่ถูกใช้ในสถานะ OPERATOR

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“=~”	เครื่องหมายเปรียบเทียบนิพจน์ปกติเป็นลักษณะที่เพิ่มเข้ามา	ไปที่สถานะ BEFORE_REG_EX และจัดเก็บสถานะ SEPARATOR
“=”	เครื่องหมายเท่ากับ	ไปที่สถานะ BEFORE_ARRAY_CLOSURE
“++”	เครื่องหมายบวก	ตรวจสอบสถานะบนสุดถ้าเป็น METHOD_STATE ให้ไปทำงานที่ METHOD_STATE แต่ถ้าไม่ใช่ให้ไปที่สถานะ LITERAL และจัดเก็บสถานะ SEPARATOR แทน
เครื่องหมายที่เหลือทั้งหมด	เครื่องหมาย	ไปที่สถานะ LITERAL และจัดเก็บสถานะ SEPARATOR
.	ไม่มี	ไปที่สถานะ LITERAL และจัดเก็บสถานะ SEPARATOR

หมายเหตุ คำสำคัญที่เหลือทั้งหมดสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.15 สถานะ CLASS_STATE (Class state)

สถานะ CLASS_STATE เป็นสถานะสุดท้ายสำหรับการตรวจสอบรูปแบบของรูปแบบการปิดประโยค โดยการตรวจสอบที่สถานะรูปแบบของการขึ้นบรรทัดใหม่จะถูกระบุให้เป็นรูปแบบการปิดประโยคเช่นเดียวกับเครื่องหมายอัฒภาค ซึ่งค่าอื่นๆ ถูกกำหนดดังตารางที่ 3.14

ตารางที่ 3.14 สัญลักษณ์ที่ถูกใช้ในสถานะ CLASS_STATE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“,”	เครื่องหมายจุลภาค	ไปที่สถานะ VARIABLE_METHOD_NAME
“.”	เครื่องหมายจุด	ไปที่สถานะ VARIABLE_METHOD_NAME และจัดเก็บสถานะ CLASS_STATE
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT
“]”	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิม

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
เครื่องหมายที่เหลือทั้งหมด	เครื่องหมาย	ไปที่สถานะ LITERAL และจัดเก็บสถานะ SEPARATOR
";"	เครื่องหมายอัฒภาค	ไปสถานะที่ CLASS_INTERFACE_BODY
"\n|\r|\r\n"	เครื่องหมายอัฒภาค	ไปสถานะที่ CLASS_INTERFACE_BODY

หมายเหตุ เครื่องหมายที่เหลือทั้งหมดสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.16 สถานะ METHOD_STATE (Method state)

เช่นเดียวกันกับสถานะ CLASS_STATE สถานะนี้ METHOD_STATE เป็นสถานะสุดท้ายสำหรับการตรวจสอบรูปแบบของรูปแบบการปิดประโยคที่เกิดในเมธอด โดยการตรวจสอบที่สถานะรูปแบบของการขึ้นบรรทัดใหม่จะถูกระบุให้เป็นรูปแบบการปิดประโยคเช่นเดียวกับเครื่องหมายอัฒภาค ซึ่งค่าอื่นๆ ถูกกำหนดดังตารางที่ 3.15

ตารางที่ 3.15 สัญลักษณ์ที่ถูกใช้ในสถานะ METHOD_STATE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
","	เครื่องหมายจุลภาค	ไปที่สถานะ VARIABLE_METHOD_NAME
"."	เครื่องหมายจุด	ไปที่สถานะ VARIABLE_METHOD_NAME
"("	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ METHOD_STATE
")"	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิม
เครื่องหมายที่เหลือทั้งหมด	เครื่องหมาย	ไปที่สถานะ LITERAL และจัดเก็บสถานะ SEPARATOR
";"	เครื่องหมายอัฒภาค	ไปสถานะที่ STATIC_METHOD_BODY
"\n|\r|\r\n"	เครื่องหมายอัฒภาค	ไปสถานะที่ STATIC_METHOD_BODY

หมายเหตุ เครื่องหมายที่เหลือทั้งหมดสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.17 สถานะ ARRAY_INIT (Array initialization state)

การสร้างอาร์เรย์แบบปกติและแบบคงที่ซึ่งสร้างปัญหาให้กับการขยายภาษานั้นจะถูกระบุความหมายที่ชัดเจน ซึ่งทำให้ปัญหาเหล่านั้นหมดไปได้ที่สถานะ ARRAY_INIT นี้โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.16

ตารางที่ 3.16 สัญลักษณ์ที่ถูกใช้ในสถานะ ARRAY_INIT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	คงอยู่ที่สถานะเดิม จัดเก็บสถานะ ARRAY_INIT
"," และ “.”	เครื่องหมายจุลภาค และเครื่องหมายจุด	คงอยู่ที่สถานะเดิม
“new”	คำสำคัญ	จัดเก็บสถานะ INIT
“null”	คำสำคัญ	คงอยู่ที่สถานะเดิม
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ ARRAY_INIT
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม
“}”	เครื่องหมายปีกกาปิด	กลับไปที่สถานะบนสุดในตัวจัดลำดับ
“)”	เครื่องหมายวงเล็บปิด	กลับไปที่สถานะบนสุดในตัวจัดลำดับ
“]”	เครื่องหมายวงเล็บปิด	กลับไปที่สถานะ BEFORE_ARGUMENT ถ้าสถานะบนสุดเป็น ARGUMENT จัดเก็บสถานะ ARRAY_INIT ถ้าไม่ใช่ไปสถานะ SEPARATOR
.	ไม่มี	ไปที่สถานะ LITERAL และจัดเก็บสถานะ ARRAY_INIT

ความสำคัญของสถานะนี้คือมีเส้นทางที่ติดต่อกับสถานะอื่นๆ เมื่อพบเครื่องหมาย “]” ซึ่งต้องพิจารณาทางเลือกของเส้นทางจากสถานะปัจจุบันร่วมกับตัวจัดเก็บสถานะ

3.1.1.18 สถานะ BEFO RE_ARGUMENT (Before argument state)

เป็นสถานะก่อนเข้าสถานะ ARGUMENT สร้างมาเพื่อรองรับปัญหาที่เกิดจากการสร้างอาร์เรย์แบบปกติและแบบคงที่ซึ่งทำงานต่อเนื่องมาจากสถานะ ARRAY_INIT โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.17

ตารางที่ 3.17 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_ARGUMENT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ไปสถานะ ARGUMENT
.	ไม่มี	ไปที่สถานะ ARGUMENT และลบสถานะบนสุดออกจากตัวจัดเก็บ

3.1.1.19 สถานะ EXCEPTION_DECL (Exception declaration state)

สถานะ EXCEPTION_DECL ออกแบบมาเพื่อรองรับรูปแบบของการประกาศการสร้างของตัวจัดการปัญหาที่มีคำสำคัญอย่าง `exception` เป็นตัวดำเนินการ โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.18

ตารางที่ 3.18 สัญลักษณ์ที่ถูกใช้ในสถานะ EXCEPTION_DECL

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม
“,” และ “.”	เครื่องหมายจุดและจุลภาค	คงอยู่ที่สถานะเดิม
“;”และ “{”	เครื่องหมายอัฒภาค และ ปีกกาเปิด	ตรวจสอบค่าในตัวจัดเก็บชั้นบนสุด ถ้าเป็น CLASS_STATE ให้ลบทิ้งและไปที่สถานะ CLASS_INTERFACE_BODY

3.1.1.20 สถานะ BEFORE_BODY (Before body state)

เนื่องจากการทำงานของลักษณะปิด (closure) ที่เพิ่มเข้ามาเป็นส่วนหนึ่งของภาษา JGroovy สถานะนี้จึงถูกสร้างขึ้นมาเพื่อรองรับไวยากรณ์ดังกล่าวโดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.19

ตารางที่ 3.19 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_BODY

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ตรวจสอบค่าในตัวจัดเก็บชั้นบนสุด ถ้าเป็น CLASS_STATE ให้ลบทิ้งและจัดเก็บสถานะ CLASS_INTERFACE_BODY และไปที่สถานะ STATIC_METHOD_BODY ถ้าเป็น METHOD_STATE ไปที่สถานะ BEFORE_CLOSURE_PARAM
“[”	เครื่องหมายวงเล็บเปิด	ไปที่สถานะ LITERAL จัดเก็บสถานะ SEPARATOR
“]”	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิม
throws"	คำสำคัญ	ไปที่สถานะ EXCEPTION_DECL
“.”	เครื่องหมายจุด	ไปที่สถานะ LOCAL_VARIABLE_NAME
“,”	เครื่องหมายอัฒภาค	ไปที่สถานะ CLASS_STATE หรือ METHOD_STATE หรือ CLOSURE_STATE ขึ้นอยู่กับค่าในตัวจัดเก็บอยู่ในตัวจัดลำดับชั้น
เครื่องหมายที่เหลือทั้งหมด	เครื่องหมาย	ไปที่สถานะ LITERAL และจัดเก็บสถานะ BEFORE_BODY
.	ไม่มี	ไปที่สถานะที่อยู่บนสุดในตัวจัดลำดับชั้น

3.1.1.21 สถานะ CONDITION (Condition state)

สถานะนี้ถูกเตรียมไว้ในลักษณะเดียวกันกับสถานะ ARGUMENT คือใช้สำหรับจัดการคำเฉพาะที่เกี่ยวข้องกับเงื่อนไขต่างๆ เช่น `if` `for` และ `while` เป็นต้น โดยตารางที่ 3.20 ได้อธิบายเครื่องหมายและคำสำคัญต่างๆ เอาไว้ดังนี้

ตารางที่ 3.20 สัญลักษณ์ที่ถูกใช้ในสถานะ CONDITION

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
คำสำคัญต่างๆ ที่ใช้เป็นเงื่อนไข	คำสำคัญ	คงอยู่ที่สถานะเดิม
“new”	คำสำคัญ	ไปที่สถานะ INIT และจัดเก็บสถานะ CONDITION และ INIT
“[”, “]”, “{” “}”, “,”	เครื่องหมายวงเล็บเปิดและปิด ปีกกาเปิดและปิด เครื่องหมายจุลภาค	คงอยู่ที่สถานะเดิม
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ CONDITION
“)”	เครื่องหมายวงเล็บปิด	กลับไปที่สถานะบนสุดในตัวจัดลำดับ
“.”และ “..”	เครื่องหมายจุด	ไปสถานะ LITERAL จัดเก็บสถานะ CONDITION
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	คงอยู่ที่สถานะเดิม
.	ไม่มี	ไปสถานะ LITERAL จัดเก็บสถานะ CONDITION

หมายเหตุ คำสำคัญต่างๆ ที่ใช้เป็นเงื่อนไขสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.22 สถานะ BEFORE_CONDITION (Before condition state)

เป็นอีกสถานะที่ถูกเตรียมไว้ในลักษณะเดียวกันกับสถานะ BEFORE_ARGUMENT เพื่อรองรับปัญหาที่เกิดจากการขยายภาษา Groovy โดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.21

ตารางที่ 3.21 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_CONDITION

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเปิด	ตรวจสอบค่าในตัวจัดเก็บชั้นบนสุด ถ้าเป็น ARGUMENT ให้ไปที่สถานะ ARGUMENT แต่ถ้าเป็น METHOD_STATEให้ไปที่สถานะ CONDITION และจัดเก็บ STATIC_METHOD_BODY ถ้าเป็น CLOSURE_STATE ให้ไปที่สถานะ CONDITION และจัดเก็บ CLOSURE
.	ไม่มี	ไปสถานะ LOCAL_VARIABLE_NAMEจัดเก็บสถานะ METHOD_STATE

3.1.1.23 สถานะ STATIC_METHOD_BODY (Static and method state)

หนึ่งในสถานะหลักเหมือนกับ CLASS_INTERFACE_BODY เพราะการทำงานภายในเมธอด และ constructor แบบคงที่นั้นเป็นส่วนหลักของโครงสร้างการเขียนโปรแกรมโดยปกติอยู่แล้ว โดยสถานะนี้เป็นส่วนหนึ่งของการจัดการปัญหาของรูปแบบการจบประโยคของไวยากรณ์ในส่วนเมธอดโดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.22

ตารางที่ 3.22 สัญลักษณ์ที่ถูกใช้ในสถานะ STATIC_METHOD_BODY

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	จัดเก็บสถานะ METHOD_STATE
"("	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ STATIC_METHOD_BODY
“for”, “while”, “if”, “switch”, “synchronized”, “catch”	คำสำคัญ	ไปสถานะ BEFORE_CONDITION จัดเก็บสถานะ METHOD_STATE
“this”, “super”		ไปสถานะ BEFORE_CONDITION จัดเก็บสถานะ ARGUMENT

สัญลักษณ์		สถานะต่อไปและลักษณะการทำงาน
“continue”, “break”	คำสำคัญ	ไปสถานะ BREAK_CONTINUE_STATE
“return”, “assert”, throw		ไปสถานะ LITERAL จัดเก็บสถานะ METHOD_STATE
“else”, “try”, “do”, “goto”, “const”, “default”, “finally”		คงอยู่ที่สถานะเดิม
“def”, “long”, “int”, “byte”, “short”, “char”, “float”, “double”, “boolean”		ไปสถานะ LOCAL_VARIABLE_NAME จัดเก็บสถานะ METHOD_STATE
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
“.”	เครื่องหมายจุด	ไปสถานะ LITERAL จัดเก็บสถานะ STATIC_METHOD_BODY
“;”	เครื่องหมายอัฒภาค	คงอยู่ที่สถานะเดิม
“}”	เครื่องหมายปีกกาปิด	กลับไปที่สถานะบนสุดในตัวจัดลำดับ
.	ไม่มี	ไปสถานะ LITERAL จัดเก็บสถานะ METHOD_STATE

สถานะ STATIC_METHOD_BODY เป็นสถานะที่มีเส้นทางติดต่อระหว่างสถานะต่างๆ หลายสถานะเพราะเป็นช่องทางหลักสำหรับส่วนการเขียนโปรแกรม

3.1.1.24 สถานะ CASE (Case state)

เป็นสถานะที่ใช้สำหรับคำสำคัญอย่าง case เนื่องจากว่ารูปแบบของ case ที่ขยายความสามารถของภาษานั้นมีเงื่อนไขที่ค่อนข้างหลากหลายและตัวภาษาหลัก

เองยังสามารถรับค่าคงที่ต่างๆ มาเพื่อเข้าเป็นเงื่อนไขได้อีกด้วย ดังนั้นในตารางที่ 3.23 จึงอธิบายสัญลักษณ์ต่างๆ ไว้ดังนี้

ตารางที่ 3.23 สัญลักษณ์ที่ถูกใช้ในสถานะ CASE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"("	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ CASE_STATE
"[", "]", ",", "...", "..", ".", ":", ";"	เครื่องหมายวงเล็บเปิดและปิด เครื่องหมายจุลภาค เครื่องหมายจุด เครื่องหมายอัฒภาค	คงอยู่ที่สถานะเดิม
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
"true", "false", "null"	คำสำคัญ	
"\""	ค่าคงที่ตริง	ไปสถานะ STRING จัดเก็บสถานะ CASE_STATE
"\'"	ค่าคงที่ตริง	ไปสถานะ GROOVY_STRING จัดเก็บสถานะ CASE_STATE
ค่าคงที่ต่างๆ	ค่าคงที่	คงอยู่ที่สถานะเดิม

หมายเหตุ ค่าคงที่ต่างๆ ที่เป็นค่าคงที่สามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.25 สถานะ LOCAL_VARIABLE_NAME (Local variable name state)

สถานะ LOCAL_VARIABLE_NAME สร้างเพื่อใช้ระบุชื่อของตัวแปรที่ถูกประกาศในระดับล่าง เช่น ในระดับเมธอด ดังนั้นที่สถานะนี้สัญลักษณ์โดยส่วนมากจะถูกส่งไปที่สถานะ SEPARATOR เพื่อใช้ระบุรูปแบบการจบประโยคแบบใช้และไม่ใช้เครื่องหมายโดยมีการกำหนดความหมายต่างๆ ดังตารางที่ 3.24

ตารางที่ 3.24 สัญลักษณ์ที่ถูกใช้ในสถานะ LOCAL_VARIABLE_NAME

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"["	เครื่องหมายวงเล็บเปิด	จัดเก็บสถานะ SEPARATOR

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT ตรวจสอบสถานะที่อยู่ด้านบนสุดในตัวจัดลำดับถ้าพบว่าเป็น CLOSURE_STATE ไม่ต้องจัดเก็บ แต่ถ้าเป็นสถานะอื่นให้จัดเก็บสถานะ BEFORE_BODY
“]”	เครื่องหมายวงเล็บปิด	ตรวจสอบสถานะบนสุดถ้าพบว่าเป็น SEPARATOR ให้ลบทิ้ง และถ้าพบว่าไม่เป็น ARRAY_INIT ให้จัดเก็บ ARRAY_INIT
“<”	เครื่องหมายน้อยกว่า	ตรวจสอบสถานะบนสุดถ้าพบว่าเป็น CLOSURE_STATE ให้ไปสถานะดังกล่าว แต่ถ้าไม่ใช่ให้ไปสถานะ GENERIC_DECL และจัดเก็บ LOCAL_VARIABLE_NAME
“.”	เครื่องหมายจุด	ตรวจสอบสถานะบนสุดถ้าสถานะบนสุดเป็น METHOD_STATE ให้ไปที่สถานะ STATIC_METHOD_BODY แต่ถ้าเป็น CLOSURE_STATE ให้ไปที่ CLOSURE
“:”	เครื่องหมายทวิภาค	ไปที่สถานะ STATIC_METHOD_BODY
“<<” และ “<<=”	คำสั่งเลื่อนไปทางซ้าย และเลื่อนไปทางซ้ายเท่ากับ	ไปที่สถานะ LITERAL จัดเก็บสถานะ SEPARATOR
คำสำคัญทั้งหมด	คำสำคัญ	ไปที่สถานะ SEPARATOR
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
.	ไม่มี	ไปสถานะ BEFORE_OPERATOR

หมายเหตุ คำสำคัญทั้งหมดสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.26 สถานะ ARGUMENT (Argument state)

สถานะ ARGUMENT เป็นสถานะปลายทางที่หลายๆ สถานะจำเป็นต้องส่งต่อค่าต่างๆ มาเพื่อระบุความหมายของคำและสัญลักษณ์ เมื่อระบุคำและสัญลักษณ์ต่างๆ ได้แล้ว สถานะ ARGUMENT จำเป็นต้องดำเนินการให้กลับไปสถานะนั้นๆ ที่ส่งผ่านมาได้อย่างถูกต้อง ดังนั้นลักษณ์การทำงานและทิศทางของค่าต่างๆ ได้ถูกระบุอยู่ในตารางที่ 3.25

ตารางที่ 3.25 สัญลักษณ์ที่ถูกใช้ในสถานะ ARGUMENT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ตรวจสอบสถานะบนสุดถ้าพบว่าเป็น INIT ให้ไปสถานะ CLASS_INTERFACE_BODY แต่ถ้าเป็นสถานะ BEFORE_BODY ให้ไปสถานะ ให้ลบทิ้ง และตรวจสอบสถานะบนสุดถ้าพบว่าเป็น ARRAY_INIT ให้ไปสถานะดังกล่าว ถ้าไม่ใช่ให้ไปสถานะ BEFORE_CLOSURE_PARAM ดูเพิ่มเติมได้ที่ภาคผนวก
“<”	เครื่องหมายน้อยกว่า	ให้ไปสถานะ GENERIC_DECL และจัดเก็บ ARGUMENT
คำสำคัญที่ระบุประเภทตัวแปร	คำสำคัญ	คงอยู่ที่สถานะเดิม
“[”	เครื่องหมายวงเล็บเปิด	ตรวจสอบสถานะบนสุดถ้าพบว่าเป็น INIT ให้ไป ARRAY_INIT และจัดเก็บสถานะ ARGUMENT
“]”	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิม
“<<” , “<<=” , “<=>”, “...”, “.”, “..”, “,”, “;”	คำสั่งเลื่อนไปทางซ้าย เลื่อนไปทางซ้ายเท่ากับ เครื่องหมายจุด เครื่องหมายจุลภาค เครื่องหมายอัฒภาค	
“(”	เครื่องหมายวงเล็บปิด	ไปสถานะที่อยู่บนสุดของตัวจัดเก็บ
“}”	เครื่องหมายปีกกาปิด	

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ตรวจสอบสถานะบนสุดถ้าพบว่าเป็น INIT ให้ไปสถานะให้จัดเก็บสถานะ ARGUMENT และ INIT ถ้าไม่ใช่ให้จัดเก็บสถานะ ARGUMENT
.	ไม่มี	ไปสถานะ LITERAL และจัดเก็บสถานะ ARGUMENT

หมายเหตุ คำสำคัญที่ระบุประเภทตัวแปรสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.27 สถานะ LITERAL (Literal state)

สถานะ LITERAL เป็นสถานะหลักที่เกี่ยวข้องและมีข้อมูลลักษณะคำต่างๆ จำนวนมากเนื่องจากค่าคงที่ทุกชนิดเช่น เลขจำนวนเต็ม ตัวอักษร เลขทศนิยม ต่างๆ ต้องถูกระบุคำจำกัดความที่สถานะนี้ อีกทั้งหลังจากได้ระบุคำจำกัดความแล้วจะต้องกลับสถานะก่อนหน้าได้อย่างถูกต้องอีกเช่นกันซึ่งค่าต่างๆ ได้ถูกระบุอยู่ในตารางที่ 3.26

ตารางที่ 3.26 สัญลักษณ์ที่ถูกใช้ในสถานะ LITERAL

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ไปสถานะ BEFORE_CLOSURE_PARAM
“[”	เครื่องหมายวงเล็บเปิด	ไปสถานะ BEFORE_ARRAY_INIT
“]”	เครื่องหมายวงเล็บปิด	ไปสถานะที่อยู่บนสุดของตัวจัดเก็บ
“(”	เครื่องหมายวงเล็บปิด	ไปสถานะ ARGUMENT
“new”	คำสำคัญ	ไปสถานะ INIT และจัดเก็บสถานะ INIT
“this”, “class”, “super”, “instanceof”, “null”, “false”, “true”		ไปสถานะที่อยู่บนสุดของตัวจัดเก็บ

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\””	ค่าคงที่สตริง	ไปสถานะ STRING จัดเก็บสถานะบนสุด
“\'”		ไปสถานะ GROOVY_STRING จัดเก็บสถานะบนสุด
“\”'”		ไปสถานะ GROOVY_PATTERN จัดเก็บสถานะบนสุด
ค่าคงที่ต่างๆ	ค่าคงที่	ไปสถานะที่อยู่บนสุดของตัวจัดเก็บ
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
เครื่องหมายดำเนินการต่างๆ	เครื่องหมาย	
.	ไม่มี	ไปที่สถานะ CLASS_STATE หรือ METHOD_STATE หรือ CLOSURE_STATE

หมายเหตุ ค่าคงที่ต่างๆ และ เครื่องหมายดำเนินการต่างๆ สามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.28 สถานะ BEFORE_REG_EX (Before regular expression state)

ในความสามารถของภาษา Groovy ที่ได้ถูกขยายลงไปในภาษา Java และสุดท้ายกลายมาเป็นภาษา JGroovy นั้นมีรูปแบบเฉพาะที่สามารถจัดการนิพจน์ปกติ (regular expression) โดยก่อนที่จะระบุนิพจน์ปกตินั้นต้องทำการตรวจสอบเงื่อนไขที่สถานะนี้ก่อน โดยมีรูปแบบต่างๆ ดังตารางที่ 3.27

ตารางที่ 3.27 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_REG_EX

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“/”	เครื่องหมายทับ	ไปสถานะ REG_EX จัดเก็บสถานะบนสุด
.	ไม่มี	ไปที่สถานะ LITERAL

3.1.1.29 สถานะ REG_EX (Regular expression state)

เป็นสถานะต่อเนื่องสำหรับอธิบายลักษณะของ นิพจน์ปกติโดยมีรูปแบบต่างๆ ดังตารางที่ 3.28

ตารางที่ 3.28 สัญลักษณ์ที่ถูกใช้ในสถานะ REG_EX

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"[^/]+"	ค่าใดๆ ที่ไม่ใช่เครื่องหมายทับ	คงอยู่ที่สถานะเดิม
"/"	เครื่องหมายทับ	ไปสถานะบนสุดของตัวจัดลำดับ

3.1.1.30 สถานะ BEFORE_OPERATOR (Before operator state)

เป็นลักษณะเดียวกับสถานะอื่นๆ ที่ต้องทำการตรวจสอบเส้นทางของค่าที่เข้ามาพร้อมกับค่าที่อยู่ในตัวจัดเก็บ ซึ่งจะใช้เป็นเงื่อนไขในการกำหนดเส้นทางของค่านั้นๆ โดยมีรูปแบบต่างๆ ดังตารางที่ 3.29

ตารางที่ 3.29 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_OPERATOR

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ดูสถานะในตัวจัดลำดับถ้าไม่พบว่ามี สถานะ METHOD_STATE ให้ระบุว่าเป็นการปิดประโยคแบบไม่ใช้เครื่องหมาย
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
";"	เครื่องหมายอัฒภาค	ดูสถานะในตัวจัดลำดับถ้าไม่พบว่ามี สถานะ METHOD_STATE ให้ระบุว่าเป็นการปิดประโยคแบบไม่ใช้เครื่องหมาย
.	ไม่มี	ไปที่สถานะ OPERATOR

3.1.1.31 สถานะ CLOSURE (Closure state)

สถานะ CLOSURE ถูกออกแบบมาเพื่อรองรับลักษณะพิเศษที่สามารถรองรับการทำงานของภาษาเฉพาะทางใดๆ ที่อยู่ภายใต้ภาษาปกติ ซึ่งโดยทั่วไปต้องสามารถรองรับการทำงานภายใต้เงื่อนไขดังสมการต่อไปนี้

$$L^* = \bigcup_{i=0}^{\infty} L^i \qquad (1)$$

กล่าวคือสถานะนี้ต้องรับอักษรใดๆ ซึ่งรูปไวยากรณ์ไม่ขัดกับโครงสร้างของภาษาหลัก ดังนั้นรูปแบบต่างๆ ในตารางที่ 3.30 อธิบายสัญลักษณ์ต่างๆ ไว้ดังนี้

ตารางที่ 3.30 สัญลักษณ์ที่ถูกใช้ในสถานะ CLOSURE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	จัดเก็บสถานะ CLOSURE
“def”, “long”, “int”, “byte”, “short”, “char”, “float”, “double”, “boolean	คำสำคัญ	ไปสถานะ LOCAL_VARIABLE_NAME จัดเก็บสถานะ CLOSURE_STATE
“for”, “while”, “if”, “switch”, “synchronized”, “catch”		ไปสถานะ BEFORE_CONDITION จัดเก็บสถานะ CLOSURE_STATE
“this”, “super”		ไปสถานะ BEFORE_CONDITION จัดเก็บสถานะ ARGUMENT
“return”		ไปสถานะ BEFORE_RETURN จัดเก็บสถานะ ARGUMENT
“assert”, “throw”		ไปสถานะ LITERAL จัดเก็บสถานะ CLOSURE_STATE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“continue”, “break”, “case”,	คำสำคัญ	ไปสถานะ LITERAL จัดเก็บสถานะ CLOSURE_STATE
“else”, “try”, “do”, “goto”, “const”, “default”, “finally”		คงอยู่ที่สถานะเดิม
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ไปสถานะ LOCAL_VARIABLE_NAME
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	จัดเก็บสถานะ CLOSURE_STATE
“:” , “;”	เครื่องหมายทวิภาค เครื่องหมายอัฒภาค	คงอยู่ที่สถานะเดิม
“.”	เครื่องหมายจุด	ถ้าสถานะบนสุดเป็นสถานะ CLOSURE_STATE ให้ลบทิ้งและคงอยู่ที่สถานะเดิม
“}”	เครื่องหมายปีกกาปิด	ไปสถานะบนสุดของตัวจัดลำดับ
.	ไม่มี	ไปที่สถานะ LITERAL NAME จัดเก็บสถานะ CLOSURE_STATE และ SEPARATOR

3.1.1.32 สถานะ CLOSURE_STATE (Closure state)

สถานะนี้ถูกกำหนดให้ทำงานในลักษณะเดียวกันกับ METHOD_STATE และสถานะ CLASS_STATE เพื่อแยกทำการแยกลักษณะการจบประโยคแบบใช้และไม่ใช้เครื่องหมายที่จะเกิดขึ้นภายในระดับชั้นการทำงานของการทำงานในรูปแบบปิด (Closure) โดยตารางที่ 3.31 ได้ทำการอธิบายสัญลักษณ์ต่างๆ ไว้ดังนี้

ตารางที่ 3.31 สัญลักษณ์ที่ถูกใช้ในสถานะ CLOSURE_STATE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"," และ "."	เครื่องหมายจุลภาค และจุด	ที่สถานะ VARIABLE_METHOD_NAME
"]"	เครื่องหมายวงเล็บปิด	คงอยู่ที่สถานะเดิม
สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"."	เครื่องหมายจุด	ไปสถานะ LOCAL_VARIABLE_NAME จัดเก็บสถานะ CLOSURE_STATE
"{"	เครื่องหมายปีกกาเปิด	ไปสถานะ BEFORE_CLOSURE_PARAM จัดเก็บสถานะ CLOSURE_STATE
"("	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ CLOSURE_STATE
เครื่องหมายที่เหลือทั้งหมด	เครื่องหมาย	ไปสถานะ LITERAL จัดเก็บสถานะ CLOSURE_STATE และ SEPARATOR
";"	เครื่องหมายอัฒภาค	ไปสถานะ CLOSURE_PARAM หรือ CLOSURE ขึ้นอยู่กับค่าที่อยู่ในตัวจัดลำดับชั้น
"}"	เครื่องหมายปีกกาเปิด	
.	ไม่มี	

หมายเหตุ เครื่องหมายที่เหลือทั้งหมดสามารถดูเพิ่มเติมได้ที่ภาคผนวก

3.1.1.33 สถานะ BEFORE_CLOSURE_PARAM (Before closure parameter state)

ถูกออกแบบมาเหมือนกันกับสถานการณ์ทำงานแบบก่อนเข้าสถานะอื่นๆ คือคอยตรวจสอบเงื่อนไขก่อนเข้าสถานะหลัก สำหรับสถานะนี้ก็เช่นเดียวกันเพราะใช้สำหรับตรวจสอบเงื่อนไขก่อนเข้าสถานะ CLOSURE_PARAM ซึ่งเป็นลักษณะการทำงานที่เพิ่มเข้ามาเนื่องจากความสามารถของภาษา Groovy ซึ่งมีเงื่อนไขที่ค่อนข้างละเอียดอ่อน โดยตารางที่ 3.32 ได้ทำการอธิบายสัญลักษณ์ต่างๆ ไว้ดังนี้

ตารางที่ 3.32 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_CLOSURE_PARAM

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“def”, “long”, “int”, “byte”, “short”, “char”, “float”, “double”, “boolean	คำสำคัญ	ไปสถานะ CLOSURE_PARAM
for”, “while”, “if”		ไปสถานะ BEFORE_CONDITION จัดเก็บสถานะ CLOSURE_STATE
“do”	คำสำคัญ	ไปสถานะ CLOSURE
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ไปสถานะ CLOSURE_PARAM_STATE_CHECK
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
.	ไม่มี	ไปสถานะ CLOSURE

3.1.1.34 สถานะ CLOSURE_PARAM (Closure parameter)

สถานะนี้เป็นหนึ่งในสถานะหลักที่ใช้ดำเนินการจัดการคัดแยกลักษณะการทำงานภายในของสถานะปิดซึ่งมีการรับ-ส่งค่าที่เป็นรูปแบบเฉพาะ โดยรูปแบบเฉพาะต่างๆ จะถูกอธิบายลักษณะไว้ในตารางที่ 3.33

ตารางที่ 3.33 สัญลักษณ์ที่ถูกใช้ในสถานะ CLOSURE_PARAM

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	ไปสถานะ CLOSURE
“def”, “long”, “int”, “byte”, “short”, “char”, “float”, “double”, “boolean	คำสำคัญ	คงอยู่ที่สถานะเดิม

3.1.1.35 สถานะ CLOSURE_PARAM_STATE (Closure parameter state)

การทำงานของสถานะ CLOSURE_PARAM_STATE นั้นจะต้องทำงานแบบต่อเนื่องร่วมกันกับสถานะ CLOSURE_PARAM เนื่องจากจะต้องคัดแยกลักษณะการทำงานภายในของสถานะปิดเช่นเดียวกันกับ CLOSURE_PARAM โดยรูปแบบเฉพาะต่างๆ จะถูกอธิบายลักษณะไว้ในตารางที่ 3.34

ตารางที่ 3.34 สัญลักษณ์ที่ถูกใช้ในสถานะ CLOSURE_PARAM

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	ไปสถานะ CLOSURE
“}”	เครื่องหมายปีกกาปิด	
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ CLOSURE_STATE และ SEPARATOR
.	ไม่มี	ไปสถานะ CLOSURE

3.1.1.36 สถานะ CLOSURE_PARAM_STATE_CHECK (Closure parameter state check)

เงื่อนไขของการทำงานในสถานะ CLOSURE โดยเฉพาะที่เป็นส่วนของการตั้งค่านั้นจะซับซ้อน เพื่อที่จะดำเนินรูปแบบให้ถูกต้องจะต้องมีสถานะที่ตรวจสอบหลายสถานะซึ่ง CLOSURE_PARAM_STATE_CHECK ก็เป็นอีกหนึ่งสถานะที่ถูกใช้ในงานลักษณะนี้ โดยรูปแบบเฉพาะต่างๆ จะถูกอธิบายลักษณะไว้ในตารางที่ 3.35

ตารางที่ 3.35 สัญลักษณ์ที่ถูกใช้ในสถานะ CLOSURE_PARAM_STATE_CHECK

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	ไปสถานะ CLOSURE
“{”	เครื่องหมายปีกกาเปิด	ไปสถานะ BEFORE_CLOSURE_PARAM
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ CLOSURE_STATE
“\"”	ค่าคงที่สตริง	ไปสถานะ LITERAL จัดเก็บสถานะ CLOSURE_STATE
“\'”		
.	ไม่มี	ไปสถานะ CLOSURE_PARAM

3.1.1.37 สถานะ BEFORE_ARRAY_CLOSURE (Before array and closure state)

ปัญหาที่เกิดจากรูปแบบของการประกาศอาร์เรย์แบบคงที่ และการทำงานแบบปิดนั้นค่อนข้างคล้ายกัน ดังนั้นสถานะนี้จะคอยกำหนดเส้นทางที่เหมาะสมโดยการตรวจสอบเงื่อนไขที่อยู่ภายในตัวจัดลำดับชั้น โดยรูปแบบเฉพาะและการทำงานต่างๆ จะถูกอธิบายลักษณะไว้ในตารางที่ 3.36

ตารางที่ 3.36 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_ARRAY_CLOSURE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"{"	เครื่องหมายปีกกาเปิด	ถ้าสถานะบนสุดเป็น ARRAY_INIT ให้ไปที่สถานะดังกล่าว ถ้าเป็น INIT ให้ไปที่ CLASS_INTERFACE_BODY แต่ถ้าเป็น METHOD_STATE หรือ CLASS_STATE ให้ไปที่ BEFORE_CLOSURE_PARAM แต่ถ้าเป็น CLOSURE_STATE ให้ไปที่ BEFORE_CLOSURE_PARAM ถ้าเป็น SEPARATOR ให้ไปที่สถานะบนสุด
.	ไม่มี	ไปสถานะ LITERAL จัดเก็บสถานะ SEPARATOR

3.1.1.38 สถานะ BEFORE_ARRAY_LIST_INIT (Before array and list initialization state)

รูปแบบที่เกิดจากรูปแบบของการประกาศอาร์เรย์แบบคงที่กับการประกาศข้อมูลที่เป็น List ก็เป็นอีกหนึ่งชนิดที่เกิดปัญหาเช่นกัน แน่นอนสถานะนี้จะคอยกำหนดเส้นทางที่เหมาะสมโดยการตรวจสอบเงื่อนไขที่อยู่ภายในตัวจัดลำดับชั้น โดยรูปแบบเฉพาะและการทำงานต่างๆ จะถูกอธิบายลักษณะไว้ในตารางที่ 3.37

ตารางที่ 3.37 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_ARRAY_LIST_INIT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ไปที่สถานะ ARRAY_INIT
“[”	เครื่องหมายวงเล็บเปิด	จัดเก็บสถานะ ARRAY_INIT
“]”	เครื่องหมายวงเล็บปิด	ถ้าสถานะบนสุดเป็น ARRAY_INIT ให้ลบทิ้ง
“;”	เครื่องหมายอัฒภาค	ไปสถานะ SEPARATOR
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ ARRAY_INIT
.	ไม่มี	ไปสถานะ ARRAY_INIT

3.1.1.39 สถานะ INIT (Variable initialization state)

สถานะ INIT ใช้สำหรับกำหนดรูปแบบการสร้างวัตถุของการเขียนโปรแกรมเชิงวัตถุซึ่งเป็นรูปแบบที่สำคัญสำหรับวิธีการเขียนโปรแกรมของภาษา Java และภาษา Groovy ซึ่งจะต้องมีการสร้างวัตถุเพื่อเป็นตัวแทนในการเรียกใช้งานของคลาส โดยรูปแบบเฉพาะและการทำงานต่างๆ จะถูกอธิบายลักษณะไว้ในตารางที่ 3.38

ตารางที่ 3.38 สัญลักษณ์ที่ถูกใช้ในสถานะ INIT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT จัดเก็บสถานะ INIT
“.”	เครื่องหมายจุด	คงอยู่ที่สถานะเดิม
“[”	เครื่องหมายวงเล็บเปิด	ลบสถานะที่อยู่บนสุดของตัวจัดลำดับ และไปสถานะ BEFORE_ARRAY_INIT
“<”	เครื่องหมายน้อยกว่า	ไปสถานะ GENERIC_DECL จัดเก็บสถานะ INIT
“long”, “int”, “byte”, “short”, “char”, “float”, “double”, “boolean	คำสำคัญ	คงอยู่ที่สถานะเดิม

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	คงอยู่ที่สถานะเดิม
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	

3.1.1.40 สถานะ STRING_REF (String reference state)

ลักษณะของการทำงานของภาษา Groovy ที่สามารถอ้างอิงถึงตัวแปรต่างๆ จากภายในสตริง ซึ่งลักษณะดังกล่าวนั้นคล้ายกับภาษา Script ต่างๆ สถานะนี้จะทำการจัดรูปแบบเพื่อระบุค่าต่างๆ ไว้ต้น โดยตารางที่ 3.39 ได้อธิบายเครื่องหมายและคำสำคัญต่างๆ เอาไว้ดังนี้

ตารางที่ 3.39 สัญลักษณ์ที่ถูกใช้ในสถานะ STRING_REF

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
ค่าสตริงใดๆ	ตัวอักษร	ไปสถานะ INSIDE_STRING_REF
“{”	เครื่องหมายปีกกาเปิด	
ช่องว่าง	ตัวอักษร	รายงานข้อผิดพลาดเพราะตัวแปรเป็นช่องว่างไม่ได้

3.1.1.41 สถานะ INSIDE_STRING_REF (Inside String reference state)

เป็นสถานะที่ทำงานต่อเนื่องจากสถานะ STRING_REF เพื่อทำการระบุค่าสามารถอ้างอิงถึงตัวแปรต่างๆ โดยตารางที่ 3.40 ได้อธิบายเครื่องหมายและคำสำคัญต่างๆ เอาไว้ดังนี้

ตารางที่ 3.40 สัญลักษณ์ที่ถูกใช้ในสถานะ INSIDE_STRING_REF

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”,“:”	เครื่องหมายปีกกาเปิด เครื่องหมายทวิภาค	ต่อสตริงจากสถานะ STRING_REF
“.”	เครื่องหมายจุด	ต่อสตริงจากสถานะ STRING_REF ไปสถานะ BEFORE_DOT_STRING_REF

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ไปสถานะ ARGUMENT_IN_STRING จัดเก็บสถานะ INSIDE_STRING_REF และต่อสตริงจากสถานะ STRING_REF
“}”, “>”	เครื่องหมายปีกกาปิด เครื่องหมายมากกว่า	ไปสถานะบนสุดในตัวจัดลำดับ และต่อสตริงจากสถานะ STRING_REF
“\””	ค่าคงที่สตริง	ไปสถานะบนสุดในตัวจัดลำดับ
ช่องว่าง	ตัวอักษร	ไปสถานะบนสุดในตัวจัดลำดับ และต่อสตริงจากสถานะ STRING_REF

3.1.1.42 สถานะ ARGUMENT_IN_STRING (Argument in String reference state)

ภายในสถานะนี้ตัวอักษรทุกๆ อย่างที่เข้ามาจะถูกนำมาต่อเป็นสตริงของตัวอักษรทั้งหมด โดยตารางที่ 3.41 ได้อธิบายเครื่องหมายและคำสำคัญต่างๆ เอาไว้ดังนี้

ตารางที่ 3.41 สัญลักษณ์ที่ถูกใช้ในสถานะ ARGUMENT_IN_STRING

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
ช่องว่าง	ตัวอักษร	ต่อสตริงจากสถานะ STRING_REF
ตัวอักขระใดๆ	ตัวอักษร	
ตัวเลขใดๆ	ตัวอักษร	
“.”	เครื่องหมายจุด	
“[”, “]”	เครื่องหมายวงเล็บเปิด ปิด	
“)”	เครื่องหมายวงเล็บปิด	ต่อสตริงจากสถานะ STRING_REF และกลับไปที่สถานะบนสุดในตัวจัดลำดับ

3.1.1.43 สถานะ BEFORE_DOT_STRING_REF (Before dot operation in string reference state)

เป็นสถานะที่ใช้กำหนดทิศทางของการรับข้อมูลเหมือนกันกับสถานะที่ทำหน้าที่ในลักษณะนี้ โดยตารางที่ 3.42 ได้อธิบายเครื่องหมายและคำสำคัญต่างๆ เอาไว้ดังนี้

ตารางที่ 3.42 สัญลักษณ์ที่ถูกใช้ในสถานะ ARGUMENT_IN_STRING

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
ตัวอักษรใดๆ	ตัวอักษร	ต่อสตริง และไปที่สถานะ DOT_STRING_REF
“@”	ตัวอักขระ	

3.1.1.44 สถานะ BEFORE_PATTERN (Before pattern state)

สถานะนี้ก็เป็นอีกสถานะหนึ่งที่ใช้กำหนดทิศทางของการรับข้อมูล โดยรูปแบบของภาษาใหม่สนับสนุนให้สามารถรับและตรวจค่าในสตริงได้ผ่านคำสั่งเฉพาะ โดยตารางที่ 3.43 ได้อธิบายเครื่องหมายและคำสำคัญต่างๆ เอาไว้ดังนี้

ตารางที่ 3.43 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_PATTERN

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“/”	เครื่องหมายทับ	ต่อสตริง และไปที่สถานะ PATTERN
.	ไม่มี	อยู่สถานะเดิม

3.1.1.45 สถานะ BEFORE_RETURN (Before return state)

รูปแบบของการกำหนดเส้นทางก่อนเข้าไปในงานของคำสั่ง return นั้น ตารางที่ 3.44 ได้อธิบายไว้ดังนี้

ตารางที่ 3.44 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_RETURN

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“{”	เครื่องหมายปีกกาเปิด	ไปที่สถานะ BEFORE_CLOSURE_PARAM

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“++”	เครื่องหมายดำเนินการ	ไปสถานะ LITERAL จัดเก็บสถานะ SEPARATOR
“--”		
.	ไม่มี	ไปสถานะ LITERAL

3.1.1.46 สถานะ PATTERN (Pattern state)

สถานะนี้เป็นสถานะหลักสำหรับกำหนดข้อมูลเพื่อมาเปรียบเทียบในรูปแบบเป็นชุดซึ่งถูกกำหนดให้เป็นนิพจน์ปกติ โดยตารางที่ 3.44 ได้กำหนดคำอธิบายไว้ดังนี้

ตารางที่ 3.45 สัญลักษณ์ที่ถูกใช้ในสถานะ PATTERN

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ต่อสตริง
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	
“/”	เครื่องหมายทับ	ต่อสตริง และไปที่สถานะบนสุดของตัวจัดลำดับ

3.1.1.47 สถานะ DOT_STRING_REF (Dot and string state)

สถานะนี้เป็นสถานะหลักเช่นเดียวกับสถานะ PATTERN โดยใช้สำหรับกำหนดข้อมูลเพื่อมาสร้างเป็นตัวแปรภายใน โดยตารางที่ 3.45 ได้กำหนดคำอธิบายไว้ดังนี้

ตารางที่ 3.46 สัญลักษณ์ที่ถูกใช้ในสถานะ PATTERN

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
ตัวอักษรใดๆ	ตัวอักษร	ต่อสตริง

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“(”	เครื่องหมายวงเล็บเปิด	ต่อสตริง และไปสถานะ LITERAL และ จัดเก็บสถานะ INSIDE_STRING_REF
“.”	เครื่องหมายจุด	ต่อสตริง และไปสถานะ BEFORE_DOT_STRING_REF
“}”	เครื่องหมายปีกกาปิด	ต่อสตริง
“/”	เครื่องหมายทับ	ไปที่สถานะบนสุดของตัวจัดลำดับ
ช่องว่าง	ตัวอักษร	ต่อสตริงและไปที่สถานะบนสุดของตัวจัดลำดับ

3.1.1.48 สถานะ STRING (String state)

สถานะที่คัดแยกข้อมูลที่เป็นสตริงแบบปกติในแบบของ Java แต่รูปแบบสตริงปกติสามารถถูกขยายออกเป็นสตริงรูปแบบต่างๆ ได้เช่นกัน โดยตารางที่ 3.46 ได้กำหนดคำอธิบายไว้ดังนี้

ตารางที่ 3.47 สัญลักษณ์ที่ถูกใช้ในสถานะ STRING

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\””	ค่าคงที่สตริง	ไปที่สถานะ BEFORE_TRIPLE_STRING
อักขระใดๆ	ตัวอักขระ	ต่อสตริง
“$”	เครื่องหมาย	
“\\b”, “\\t”, “\\$”, “\\n”, “\\f”, “\\r”, “\\\”,“\\'”, “\\\\”,	ตัวอักขระ	
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	

3.1.1.49 สถานะ BEFORE_ARRAY_INIT (Before array initialization state)

เป็นลักษณะเดียวกับสถานะอื่นๆ ที่ต้องทำการตรวจสอบเส้นทางของค่าที่เข้ามาพร้อมกับค่าที่อยู่ในตัวจัดเก็บ ซึ่งจะใช้เป็นเงื่อนไขในการกำหนดเส้นทางของค่านั้นๆ โดยมีรูปแบบต่างๆ ดังตารางที่ 3.49

ตารางที่ 3.48 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_ARRAY_INIT

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"]"	เครื่องหมายวงเล็บปิด	ไปที่สถานะบนสุดของตัวจัดลำดับ
.	ไม่มี	ไปสถานะ BEFORE_ARRAY_LIST_INIT

3.1.1.50 สถานะ BEFORE_TRIPLE_STRING (Before triple string state)

เป็นสถานะรูปแบบเดียวกันกับ BEFORE_ARRAY_INIT คือต้องดูเงื่อนไขในตัวจัดลำดับชั้นเป็นหลักโดยมีรูปแบบต่างๆ ดังตารางที่ 3.49

ตารางที่ 3.49 สัญลักษณ์ที่ถูกใช้ในสถานะ BEFORE_TRIPLE_STRING.

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"\""	ค่าคงที่สตริง	ไปที่สถานะ TRIPLE_STRING
.	ไม่มี	ไปสถานะ BEFORE_ARRAY_LIST_INIT

3.1.1.51 สถานะ TRIPLE_STRING (Triple string state)

เป็นสถานะหลักที่ใช้ระบุสตริงที่เป็นรูปแบบพิเศษซึ่งเป็นลักษณะของภาษา Groovy ที่คล้ายคลึงกับภาษา Script โดยรูปแบบต่างๆ ที่ใช้ในการระบุนั้นถูกกำหนดอยู่ในตารางที่ 3.50

ตารางที่ 3.50 สัญลักษณ์ที่ถูกใช้ในสถานะ TRIPLE_STRING

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
"\"\"\""	ค่าคงที่สตริง	ไปที่สถานะบนสุดของตัวจัดลำดับ
อักขระใดๆ	ตัวอักขระ	ต่อสตริง

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\””	ค่าคงที่สตริง	ต่อสตริง
“\\b”, “\\t”, “\\$”, “\\n”, “\\f”, “\\r”, “\\\”,“\\'”, “\\\\”,	ตัวอักขระ	
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	

3.1.1.52 สถานะ BREAK_CONTINUE_STATE (Break and continue state)

สถานะ BREAK_CONTINUE_STATE เป็นสถานะที่ระบุทางเลือกของคำสำคัญที่เกี่ยวข้องเพื่อที่จะระบุสถานะถัดไปอย่างเหมาะสม โดยมีรูปแบบต่างๆ ดังตารางที่ 3.51

ตารางที่ 3.51 สัญลักษณ์ที่ถูกใช้ในสถานะ BREAK_CONTINUE_STATE

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	ไปสถานะ STATIC_METHOD_BODY
“;”	เครื่องหมายอัฒภาค	
.	ไม่มี	ไปสถานะ LITERAL และจัดเก็บสถานะ BREAK_CONTINUE_STATE

3.1.1.53 สถานะ GROOVY_PATTERN (Groovy string pattern state)

สถานะนี้เป็นสถานะหลักสำหรับกำหนดข้อมูลเพื่อมาเปรียบเทียบในรูปแบบเป็นชุดซึ่งถูกกำหนดให้เป็นนิพจน์ปกติเช่นเดียวกับสถานะ PATTERN โดยตารางที่ 3.53 ได้กำหนดคำอธิบายไว้ดังนี้

ตารางที่ 3.52 สัญลักษณ์ที่ถูกใช้ในสถานะ GROOVY_PATTERN

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	ต่อสตริง
ช่องว่าง	ค่าคงที่สตริง	
ตัวอักษร	ตัวอักษร	
“\””	ค่าคงที่สตริง	ไปที่สถานะบนสุดของตัวจัดลำดับ

3.1.1.54 สถานะ GROOVY_STRING (Groovy string state)

สถานะที่คัดแยกข้อมูลที่เป็นสตริงแบบปกติในแบบของ Java ดังที่ได้อธิบายไปในสถานะ STRING แต่รูปแบบสตริงปกติสามารถถูกขยายออกเป็นสตริงรูปแบบต่างๆ ได้เช่นกันด้วยลักษณะการทำงานของภาษา Groovy โดยตารางที่ 3.53 ได้กำหนดคำอธิบายไว้ดังนี้

ตารางที่ 3.53 สัญลักษณ์ที่ถูกใช้ในสถานะ GROOVY_PATTERN

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
“\'”	ค่าคงที่สตริง	ไปที่สถานะบนสุดของตัวจัดลำดับ
ตัวอักษร	ตัวอักษร	ต่อสตริง
“\\b”, “\\t”, “\\$”, “\\n”, “\\f”, “\\r”, “\\\”,“\\'”, “\\\\”,	ตัวอักขระ	
“\n|\r|\r\n”	สิ้นสุดการทำงานของประโยค	

3.1.1.55 สถานะ CLOSURE_PARAM_METHOD (Closure parameter method invocation state)

ลักษณะของการเรียกใช้เมธอดส่วนรับค่าของสถานะปิดมีลักษณะที่แตกต่างจากการเรียกใช้เมธอดทั่วไป โดยสถานะนี้ต้องกำหนดทางเลือกที่เหมาะสมให้กับค่าที่เข้ามา โดยตารางที่ 3.54 ได้กำหนดคำอธิบายไว้ดังนี้

ตารางที่ 3.54 สัญลักษณ์ที่ถูกใช้ในสถานะ CLOSURE_PARAM_METHOD

สัญลักษณ์	ชนิดของสัญลักษณ์	สถานะต่อไปและลักษณะการทำงาน
คำสำคัญต่างๆ	คำสำคัญ	ไปที่สถานะ CLOSURE
":" , ";", "." , "}"	เครื่องหมายทวิภาค เครื่องหมายอัฒภาค เครื่องหมายจุด เครื่องหมายปีกกาปิด	
Uppercase Identifier	ตัวอักษรตัวแรกที่เป็นตัวใหญ่	ไปที่สถานะ LITERAL
Lowercase Identifier	ตัวอักษรตัวแรกที่เป็นเล็ก	

3.1.1.56 สถานะ DEFAULT (Default state)

เป็นสถานะสุดท้ายที่คอยรับข้อผิดพลาดจากการคัดแยกสตริงจากสถานะ STRING GROOVY STRING TRIPLE_STRING REG_EX และ STRING_REF โดยจะแสดงข้อผิดพลาดเมื่อมีการออกแบบสถานะใดสถานะหนึ่ง

โดยทั่วไปเส้นทางของสถานะต่างๆ นั้นโดยส่วนมากแล้ววงเล็บเปิดและปิดนั้นเป็นเครื่องหมายหลักที่ใช้ในดำเนินการพิจารณาทางเลือก ซึ่งการพิจารณาทางเลือกนั้นจะต้องมีเงื่อนไขของสถานะที่อยู่ในตัวจัดเก็บลำดับชั้นมาเป็นส่วนประกอบในการพิจารณาสร้างทางเลือกให้เหมาะสมกับค่านั้นๆ และค่าต่อไปเป็นสำคัญ

3.3.2 ตัวตรวจสอบไวยากรณ์ (Syntactic Analyzer)

ไวยากรณ์ของภาษานั้นถูกออกแบบด้วยลักษณะไวยากรณ์ข้อมูล (Attribute grammar) ซึ่งจะรับคำที่ผ่านส่วนของการคัดแยกคำแล้วมาเปรียบเทียบกับไวยากรณ์ของภาษา โดยไวยากรณ์ของภาษา Java5 นั้นถูกระบุอยู่ที่ JLS [19] ซึ่งรูปแบบที่ชัดเจนเช่นเดียวกับกับไวยากรณ์ของภาษา Groovy เมื่อนำไวยากรณ์ของภาษาทั้ง 2 ชนิดเข้ามารวมกันจะได้ไวยากรณ์ของภาษา JGroovy

การกำหนดไวยากรณ์ของภาษาจะต้องออกแบบและกำหนดไวยากรณ์ของภาษาหลักก่อน ดังนั้นไวยากรณ์ของภาษา Java จะถูกระบุเป็นตัวแรกโดยรูปแบบของไวยากรณ์เป็นดังนี้

- ต้องออกแบบโครงสร้างต้นไม้เชิงนามธรรม [24] (AST) ก่อน
- ออกแบบไวยากรณ์ให้ค่าไวยากรณ์ข้อมูลสอดคล้องกับสมาชิกของข้อมูลภายในของโครงสร้างต้นไม้เชิงนามธรรมนั้นๆ

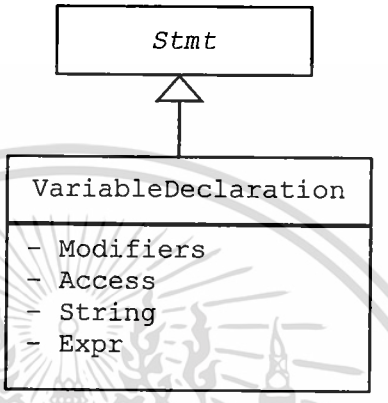
ยกตัวอย่างโครงสร้างไวยากรณ์ของการประกาศตัวแปร

```
int sample = 0 ;
```

จะมีโครงสร้างของโครงสร้างต้นไม้เชิงนามธรรมเป็นดังนี้

```
VariableDeclaration : Stmt ::= Modifiers TypeAccess:Access
                               <ID:String> [Init:Expr];
```

โดยสามารถเขียนออกมาเป็นโครงสร้างคลาสดังรูปที่ 3.1



รูปที่ 3.1 โครงสร้างคลาส Stmt

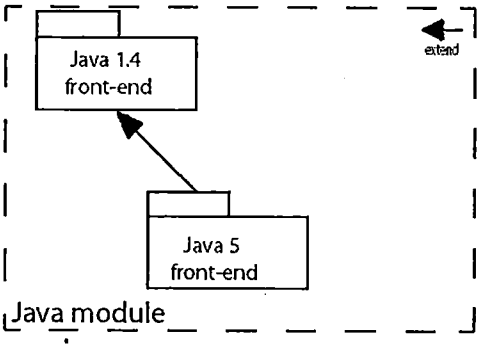
โดยไวยากรณ์จะสามารถกำหนดออกมาให้เหมาะสมกับรูปแบบของการประกาศตัวแปรและโครงสร้างของโครงสร้างต้นไม้เชิงนามธรรมดังตัวอย่างข้างล่าง

```
BodyDecl field_declaration = modifiers.m? type.t
                             variable_declarators.v SEMICOLON
```

ซึ่งจะยึดกฎทางขวา (Right hand side rule) โดยมีสถานะไม่สิ้นสุดอยู่ทางซ้าย (Nonterminal) และสถานะสิ้นสุดอยู่ทางขวา (Terminal) [56] โดยทางค่า `variable_declarators.v` จะสามารถมีกฎต่อไปได้อีกคือ

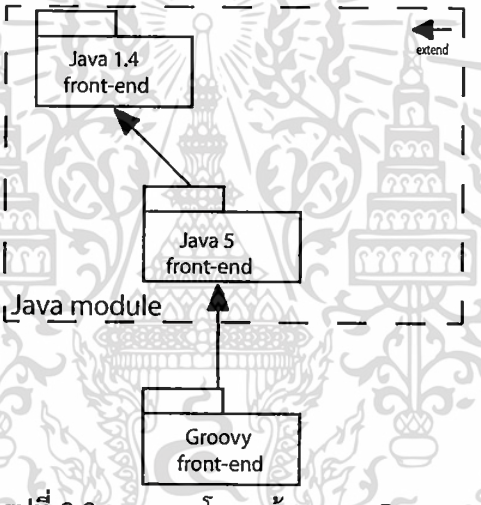
```
VariableDecl variable_declarator =
               variable_declarator_id.
           |   variable_declarator_id.v EQ variable_initializer.i
```

จะเห็นว่าจำนวนสมาชิกของคลาสจะสอดคล้องกับไวยากรณ์ของภาษา โดยการสร้างไวยากรณ์ในบางโครงสร้างจะมีการใช้สมาชิกที่มีสถานะสิ้นสุดร่วมกันได้ [57] แต่ท้ายสุดแล้วไวยากรณ์ที่สมบูรณ์จะต้องไม่มีโครงสร้างเหมือนกันทั้งหมดเพราะจะทำให้ภาษาเกิดความสับสนและกำกวม ไวยากรณ์ของภาษา Java จะถูกออกแบบก่อนและรวมกันอยู่ในส่วนเดียวกันดังรูปที่ 3.2



รูปที่ 3.2 การขยายโครงสร้างภาษา Java5

ส่วนในไวยากรณ์ของภาษาที่ถูกนำมาขยายเพิ่มลงไปในภายหลังนั้นจะขยายต่อจากส่วนสุดท้ายของภาษา Java5 การขยายในลักษณะนี้ตัวสร้างตัวแปลภาษาจะทำการตรวจรูปแบบไวยากรณ์ที่ถูกกำหนดขึ้นโดยเริ่มจากภาษาหลักคือ Java1.4 มาที่ Java5 และส่วนของภาษา Groovy ตามลำดับดังรูปที่ 3.3



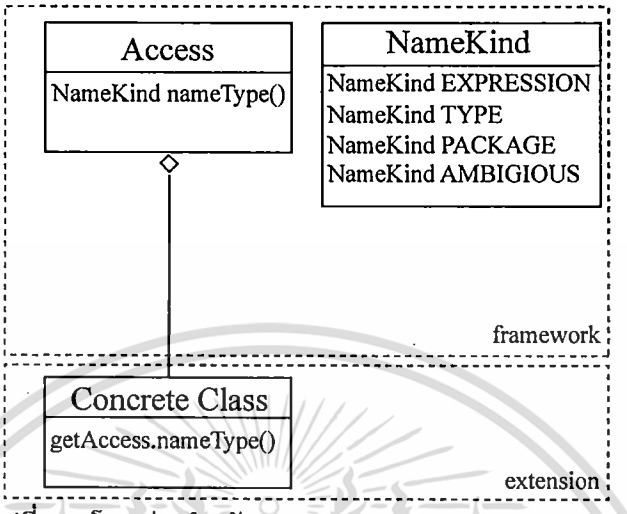
รูปที่ 3.3 การขยายโครงสร้างภาษา Groovy

ซึ่งจะมาการสร้างโครงสร้างต้นไม้เชิงนามธรรมออกมาเพื่อรองรับไวยากรณ์ของภาษา Groovy แต่การสร้างไวยากรณ์บางส่วนอาจจะมีการนำโครงสร้างเก่ามาขยาย หรือนำมาใช้ซ้ำได้ โดยสุดท้ายโครงสร้างของไวยากรณ์ทั้งหมดจะต้องสอดคล้องกัน [58]

3.3.3 ตัวตรวจสอบความหมายภาษา (Semantic Analyzer)

เนื่องจากการสร้างไวยากรณ์ของภาษามีการใช้โครงสร้างของโครงสร้างต้นไม้เชิงนามธรรมที่เป็นคลาสเชิงวัตถุ ซึ่งจะแบ่งลำดับชั้นการทำงานต่างๆ ออกมาเป็นส่วนๆ โดยแต่ละส่วนจะมีหน้าที่ตรวจสอบความหมายของค่าที่ถูกจัดเก็บในคลาสในรูปของคุณสมบัติของคลาสอยู่แล้ว ซึ่งค่าต่างๆ จะถูกจัดเก็บอยู่ในพื้นที่กลางเพื่อเรียกค่าแต่ละตัวมาทำการตรวจความหมาย

โครงสร้างของคลาสแต่ละชนิดจะประกอบด้วยส่วนการทำงานที่เหมือนๆ กันอยู่ 2 ส่วนใหญ่ๆ คือเมธอด `nameType` และเมธอด `nameCheck` ซึ่งจะคอยตรวจสอบความถูกต้องของความหมายของภาษาในส่วนสำคัญ [] โดยใช้ลักษณะโครงสร้างการตรวจสอบดังรูปที่ 3.4

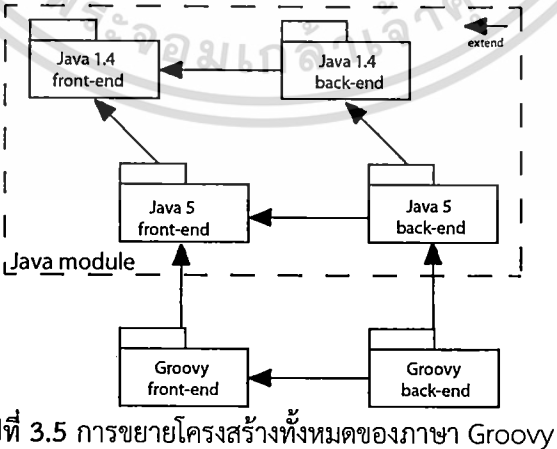


รูปที่ 3.4 โครงร่างสำหรับการตรวจสอบความหมายของภาษา

โดยโครงสร้างดังกล่าวเมื่อทำงานร่วมกับรูปแบบการเยี่ยมชม [59] (Visitor pattern) จะทำให้การทำงานของส่วนการตรวจสอบความหมายของภาษาสมบูรณ์

3.4 ปัญหาและการแก้ไขปัญหาในส่วนสร้างไบต์โค้ดของภาษา JGroovy

ส่วนของการแปลรหัสต้นฉบับให้ไปเป็นไบต์โค้ดนั้นเป็นหน้าที่ของตัวแปลภาษาส่วนหลัง (back end) โดยปกติแล้วการสร้างตัวแปลภาษาส่วนหลังต้องทำหลังจากออกแบบและสร้างตัวแปลภาษาส่วนหน้าเสร็จสิ้นแล้ว ซึ่งหมายความว่าการสร้างตัวแปลภาษาส่วนหลังต้องขยายออกมาจากตัวแปลภาษาส่วนหน้า และส่วนต่อขยายของภาษาก็จะต้องขยายต่อออกมาอีกดังรูปที่ 3.5



รูปที่ 3.5 การขยายโครงสร้างทั้งหมดของภาษา Groovy

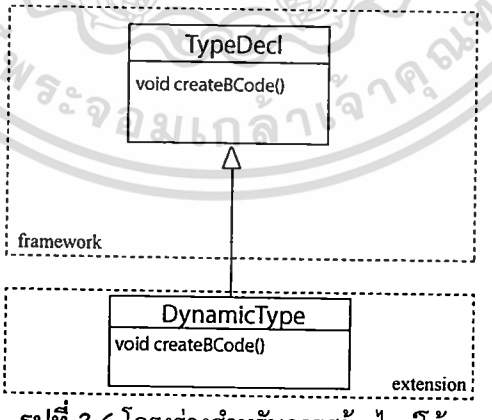
แต่เนื่องจากภาษา JGroovy นั้นมีส่วนที่เป็นการทำงานแบบพลวัตรมาจากภาษา Groovy แต่การจะสร้างไบต์โค้ดโดยปกติจะอิงจากภาษา Java เป็นหลักเนื่องจากเป็นภาษาหลัก ดังนั้นการสร้างไบต์โค้ดแบบปกติจะไม่สามารถสนับสนุนการทำงานของภาษา Groovy ที่ถูกนำมาขยายความสามารถลงไปได้

โดยปกติตัวแปลภาษาของภาษา Groovy จะทำการสร้างเมธอดพิเศษเพื่อสนับสนุนการทำงานแบบ Meta Object Protocol (MOP) สำหรับการทำงานให้เข้ากับ API ของภาษา Groovy โดยทั่วไปการตรวจสอบตัวแปรและเมธอดต่างๆ รวมไปถึงการสร้างไบต์โค้ดนั้นโครงสร้างในส่วนตรวจสอบความหมายของภาษาและตัวแปลภาษาส่วนหลังจะเป็นตัวดำเนินการ แต่รูปแบบพิเศษนี้จะไม่ปรากฏอยู่ในโค้ดทั่วไปตั้งแต่แรก ทำให้ไม่สามารถถูกตรวจสอบและดำเนินการแปลได้

ลักษณะการทำงานของภาษา Groovy นั้นถูกออกแบบมาเพื่อ API ของภาษา Groovy ที่เรียกว่า Groovy API โดยเฉพาะแน่นอนว่าหากนำลักษณะวัตถุเข้าไปทำงานร่วมกับ Groovy API หรือนำ Groovy API มาใช้โดยภาษา Java ปกตินั้นสามารถทำได้เพราะเนื่องจาก Groovy API สามารถทำงานได้บน JVM หรืออีกกรณีคือแปลรหัสต้นฉบับให้เป็นแบบ Groovy ทั้งหมดก็สามารถทำงานบน JVM ได้เช่นเดียวกัน แต่การทำในลักษณะนี้จะเป็นการเพิ่มภาระงานเพราะถ้าไม่มีการเรียกใช้งาน Groovy API ก็จะมีการสร้างเมธอดออกมาอยู่ดังนั้นการทำงานของตัวแปลภาษา JGroovy จึงต้องมีลักษณะการทำงาน 2 ลักษณะคือทำงานเพื่อสนับสนุนการแปลภาษา Java แบบปกติ และการทำงานแบบสนับสนุนการแปลทั้ง 2 ภาษา โดยการทำงานของตัวแปลภาษาส่วนหลังของ JGroovy มีลักษณะเฉพาะ

3.4.1 ตัวแปลภาษาเพื่อรองรับภาษา Java (Java Code generation)

โครงสร้างของโครงสร้างต้นไม้เชิงนามธรรมนอกจากจะเก็บข้อมูลสำหรับตรวจสอบความหมายของภาษาแล้ว ภายในคลาสแต่ละตัวจะมีส่วนๆ หนึ่งที่เป็นเมธอดสำหรับการสร้างไบต์โค้ดฝังอยู่ ซึ่งหมายความว่าการแปลข้อมูลให้เป็นไบต์โค้ดนั้นทุกๆ คลาสจะมีหน้าที่แปลในส่วนที่ตัวเองเกี่ยวข้อง โดยมีโครงสร้างการแปลดังรูปที่ 3.6



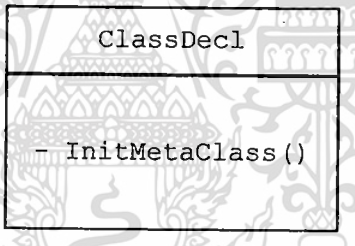
รูปที่ 3.6 โครงร่างสำหรับการสร้างไบต์โค้ด

การแปลภาษานั้นจะทำงานในลักษณะเดียวกันกับการตรวจสอบความหมายคือข้อมูลของวัตถุทุกๆ ชนิดจะถูกเก็บอยู่ที่ๆ เดียวกันเรียกว่าสระข้อมูลกลาง (Common instance pool) และเมื่อเข้ามาทำงานในส่วนของการแปลทุกๆ ข้อมูลจะถูกดึงออกมาทำงานร่วมกับรูปแบบการเยี่ยมชมเพื่อแปลงข้อมูลที่ผ่านการตรวจสอบเหล่านั้นให้ไปเป็นไบต์โค้ด ซึ่งตรงไปตรงมาถ้าเป็นการแปลภาษา Java ปกติ

3.4.2 ตัวแปลภาษาเพื่อรองรับภาษา Groovy (Groovy Code generation)

การที่จะทำการแปลรหัสต้นฉบับนั้นข้อมูลทุกอย่างจะต้องถูกเก็บอยู่ในสระข้อมูลกลาง ซึ่งนั่นจะทำให้เกิดปัญหาในการแปลภาษาในส่วนนี้เพราะรหัสต้นฉบับไม่ได้มีโค้ดพิเศษปรากฏขึ้น อีกทั้งการแปลภาษา Java ทั่วไปจะไม่จำเป็นต้องสร้างโค้ดพิเศษลงไปในไบต์โค้ดทำให้เกิดข้อแตกต่างระหว่างการแปลภาษาทั้ง 2 ชนิด

จากโครงสร้างของการแปลภาษาพบว่าการแปลภาษานั้นจะทำการแปลภาษาที่ละคลาสไฟล์ โดยเริ่มต้นจากคลาส ตัวแปรระดับสูง เมธอด และตัวแปลระดับล่าง ตามลำดับ ซึ่งแน่นอนว่าข้อมูลทั้งหมดจะถูกเก็บอยู่ที่สระข้อมูลกลาง ดังนั้นการแปลภาษาจะใช้ประโยชน์จากรูปแบบเดิมด้วยการสร้างเมธอด `initMetaClass` ขึ้นมา (ดังรูปที่ 3.7) เพื่อจัดการสร้างโค้ดพิเศษที่เรียกว่ารหัสจำลอง (Dummy code) เมื่อพบว่ามีการเรียกใช้งานที่เกี่ยวข้องกับ Groovy API



รูปที่ 3.7 เมธอดสำหรับสนับสนุนการสร้างรหัสจำลอง

3.4.1.1 การฉีดรหัสจำลองเข้าสู่รหัสหลัก (Dummy code injection)

การสร้างรหัสจำลองนั้นเมื่อได้สร้างเสร็จแล้วข้อมูลของรหัสจำลองจะอยู่นอกสระข้อมูลกลางซึ่งหมายความว่ารหัสจำลองจะไม่ถูกทำการแปลให้เป็นไบต์โค้ด ดังนั้นจะต้องทำการแทรกรหัสจำลองให้ไปรวมอยู่กับสระข้อมูลกลาง เพื่อที่จะได้ดำเนินการแปลภาษาตามขั้นตอนปกติ

การฉีดข้อมูลของรหัสจำลองจะเริ่มในช่วงของ Runtime โดยเมื่อมีการดึงข้อมูลจากสระข้อมูลกลางออกมาเพื่อสร้างเป็นข้อมูลรวม ช่วงเวลานี้จะมีการเรียกคลาสไฟล์ขึ้นมาตามลำดับที่จัดเก็บซึ่งข้อมูลจะอยู่ในรูปแบบของคลาสเชิงวัตถุอยู่แล้ว ดังนั้นเมื่อตรวจสอบว่าคลาสลำดับนั้นมีการสร้างรหัสจำลองขึ้นก็จะให้แทรกรหัสจำลองเข้าไปในส่วนรหัสหลักของคลาสนั้นๆ

เมื่อได้แทรกข้อมูลลงไปในรหัสหลักได้แล้วตัวแปลภาษาส่วนหลังก็จะทำการแปลออกมาเป็นไบต์โค้ดที่สนับสนุนการทำงานของ Groovy API [9]

3.4.3 การเพิ่มประสิทธิภาพของไบต์โค้ด (Byte code optimization)

โดยปกติแล้วการเพิ่มประสิทธิภาพของไบต์โค้ดจะทำการดัดแปลงไบต์โค้ดเดิมให้มีความสามารถในการจัดการคำสั่งต่างๆ ให้เหมาะสมยิ่งขึ้น โดยการทำงานในส่วนนี้ในปัจจุบันได้มีงานวิจัยและโครงร่างจำนวนมากอย่างเช่นโครงร่าง Soot [60] ทางตัวแปลภาษาเองได้ทำการสร้างส่วนเชื่อมต่อภายในเพื่อส่งไบต์โค้ดไปทำงานร่วมกับโครงร่างต่างๆ [61] โดยสามารถทำงานร่วมกันได้เป็นอย่างดี

บทที่ 4
การทดลองและผลการทดลองการทำงานของตัวแปลภาษา

ตัวแปลภาษาเป็นเครื่องมือสำหรับแปลรหัสต้นฉบับให้เป็นรหัสที่เหมาะสมกับเครื่องเป้าหมาย (Target machine code) โดยตัวแปลภาษาจะต้องสนับสนุนโครงสร้างที่เป็นลักษณะเฉพาะคือ คำสำคัญ ไวยากรณ์ และความหมายของภาษานั้นๆ โดยตัวแปลภาษา JGroovy (JGroovy compiler, JGC) จะต้องสนับสนุนภาษาหลักคือภาษา Java [41, 62] และภาษาที่ถูกขยายความสามารถลงไปในภาษาหลักอย่างภาษา Groovy ซึ่งอย่างน้อยจะต้องสามารถทำการแปลรหัสได้ไม่น้อยกว่าตัวแปลภาษาที่เป็นทางการอย่าง Javac ซึ่งพัฒนาโดยบริษัท Sun Microsystem [63] และตัวแปลภาษาที่เป็นทางการของภาษา Groovy ที่สามารถหาได้จากเว็บทางการของ Groovy [9]

โดยทั่วไปแล้วการทดสอบจะต้องใช้กรณีหรือชุดทดสอบที่ไม่ทำให้เกิดการได้เปรียบ หรือเสียเปรียบระหว่างตัวทดสอบโดยจะใช้ชุดทดสอบเดียวกันในสภาพแวดล้อมและเงื่อนไขที่เหมือนกัน การทดสอบนี้จะเตรียมชุดทดสอบเพื่อประเมินการทำงานของตัวแปลภาษาในรูปแบบต่างๆ ขึ้นอยู่กับวัตถุประสงค์ของการทดสอบ

ในบทนี้ได้แสดงผลการทดสอบตัวแปลภาษา JGC ด้วยชุดทดสอบต่างๆ โดยวิธีการนั้นใช้วิธีการต่างๆ หลายวิธีการอย่างเช่น ชุดทดสอบของ Jacks [64] ตัววัดประสิทธิภาพของการทำงานช่วงทำงานจริงที่เรียกว่า Grande Benchmark [65, 66] และเครื่องมือวัดการใช้ทรัพยากรเครื่องที่เรียกว่า VisualVM [67] เป็นต้น โดยการทดสอบนี้มุ่งเน้นเพื่อประเมินผลการทำงานของตัวแปลภาษา JGC ซึ่งมีผลต่อการทำงานของกับเครื่องคอมพิวเตอร์ โดยการทดสอบนี้จะทดสอบทั้งหมด 4 ส่วนด้วยกันคือ

- ทดสอบการทำงานของตัวแปลภาษาส่วนหน้าซึ่งทำการตรวจสอบภาษาหลักคือภาษา Java
- ทดสอบการทำงานของตัวแปลภาษาส่วนหน้าซึ่งทำการตรวจสอบภาษาที่ถูกขยายลงไปคือภาษา Groovy
- ทดสอบการแปลงรหัสต้นฉบับให้เป็นไบต์โค้ด
- ทดสอบประสิทธิภาพการทำงานของไบต์โค้ดในการทำงานที่แตกต่างกัน

4.1 การทดสอบการทำงานของตัวแปลภาษาส่วนหน้าของภาษา Java

การทดสอบในตัวแปลภาษาส่วนหน้านั้นถูกทดสอบด้วยเครื่องคอมพิวเตอร์ของ Asus รุ่น A8jseries Duo T2350 โดยมีหน่วยความจำ 1 กิกะไบต์ และทดสอบบนระบบปฏิบัติการ Microsoft Window Xp Sp2 เพื่อทดสอบผลกระทบต่อภาษาหลักอย่างภาษา Java จึงได้นำตัวแปลภาษาที่สามารถทำงานกับภาษา Java โดยเฉพาะอย่างตัวแปลภาษา Java ตัวแปลภาษา JastAddJ5 มาเปรียบเทียบกับตัวแปลภาษา JGC โดยใช้ชุดทดสอบของ Jacks มาทดสอบซึ่งประกอบไปด้วยกรณีทดสอบทั้งหมด 4619 กรณี โดยแต่ละตัวแปลภาษาจะถูกทดสอบ 3 ครั้งและนำมาหาค่าเฉลี่ยของการทดสอบ ชุดทดสอบของ Jacks นั้นถูกสร้างและจัดการภายใต้โครงงานของ Mauve ซึ่งอ้างอิงเพื่อออกแบบกรณีทดสอบจากมาตรฐานของภาษา Java

ชุดทดสอบของ Jacks จะประกอบไปด้วยส่วนต่างๆ ที่เป็นกรณีทดสอบโดยสามารถแยกเป็นส่วนหลักๆ ออกมาเป็นดังต่อไปนี้

- Arrays
- Blocks and statements
- Classes
- Conversions and promotions
- Definite assignment
- Expressions
- Interfaces
- Lexical structure
- Names
- Packages
- Types values and variables
- Class file format

โดยกรณีทดสอบแต่ละชุดจะถูกอธิบายในหัวข้อย่อยภายใต้หัวข้อนี้

4.1.1 การทดสอบไวยากรณ์ของอาร์เรย์ (An experiment of arrays)

สำหรับการทดสอบไวยากรณ์ของอาร์เรย์ชุดทดสอบนั้นได้นำมาตรฐานจาก JLS มาทำการสร้างชุดทดสอบ โดยกรณีทดสอบจะแสดงผลลัพธ์เมื่อทำการอ่านข้อมูลขอกรณีทดสอบนั้นๆ อย่างเช่นหนึ่งในกรณีทดสอบของอาร์เรย์ต่อไปนี้

```
tcltest::test 10.6-syntax-6 { array initializers may nest } {
    empty_main T106i6 {
        int[][] iaa = { {1}, {2} };
    }
} PASS
```

โดยกรณีทดสอบนี้เป็นกรณีทดสอบเพื่อกำหนดค่าตั้งต้นของอาร์เรย์ชนิดตัวเลขซึ่งสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
public class T106i6 {
    T106i6 (){}
    public static void main(String[] args) {
        int[][] iaa = { {1}, {2} };
    }
}
```

กรณีทดสอบครั้งนี้จะถูกรวมไว้ในไฟล์เดียวและ Jacks จะทำการส่งกรณีทดสอบทีละกรณีเข้าไปเพื่อทดสอบตัวแปลภาษา โดยในการทดสอบอาร์เรย์นี้มีกรณีทดสอบทั้งหมด 19 กรณีและทำการทดสอบตัวแปลภาษาทั้ง 3 ตัวคือ Javac JastAddJ5 และ JGC โดยผลการทดสอบถูกแสดงไว้ในตารางที่ 4.1

ตารางที่ 4.1 แสดงผลการทดสอบของอาร์เรย์

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	18	1	94.73
JastAddJ5	19	0	100
JGC	19	0	100

โดยในหัวข้อย่อยต่อไปจะเป็นการทดสอบของชุดสอบ Block-and-Statement โดยชุดทดสอบเหล่านี้เป็นการประเมินการทำงานของตัวแปลภาษาเพื่อรับรองว่าจะสามารถทำงานได้ถูกต้องตามเงื่อนไขที่ระบุไว้

4.1.2 การทดสอบไวยากรณ์ของช่องและรูปประโยค (An experiment of blocks and statements)

ชุดของกรณีทดสอบนี้ถูกสร้างขึ้นมาเพื่อตรวจสอบความถูกต้องของไวยากรณ์ต่างๆ ที่เกี่ยวข้องกับช่องและรูปประโยค ซึ่งจะประกอบไปด้วยคำสั่งต่างๆ เช่น break statement continue statement และ if statement เป็นต้น โดยสามารถแยกเป็นหัวข้อย่อยๆ ดังนี้

4.1.1.1 Break statement

ชุดกรณีทดสอบของ break statement ถูกออกแบบมาหลายๆ กรณีในหลายเงื่อนไข โดยมีกรณีทดสอบทั้งหมด 28 กรณี ซึ่งสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
public class T1414a1 {
    T1414a1 (){}
    public static void main(String[] args) {

        a: do {
            break 1;
        } while (false);
    }
}
```

ผลการทดสอบชุดกรณีทดสอบของ break statement ทั้งหมดถูกแสดงอยู่ในตารางที่ 4.2 โดยมีผลเป็นดังนี้

ตารางที่ 4.2 แสดงผลการทดสอบของ Break statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	28	0	100
JastAddJ5	28	0	100
JGC	28	0	100

4.1.1.2 Continue statement

กรณีทดสอบของ Continue statement ถูกออกแบบมาเพื่อตรวจสอบความถูกต้องของการทำงานของตัวแปลภาษา โดยใช้มาตรฐานจาก JLS ซึ่งสามารถสร้างกรณีทดสอบออกมาได้ทั้งหมด 27 กรณี โดยสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
public class T1415a1 {
    T1415a1 (){}
    public static void main(String[] args) {

        a: do {
            continue 1;
        } while (false);

    }
}
```

และผลการทดสอบชุดกรณีทดสอบของ Continue statement ทั้งหมดถูกแสดงอยู่ในตารางที่ 4.3 โดยมีผลเป็นดังนี้

ตารางที่ 4.3 แสดงผลการทดสอบของ Continue statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	28	0	100
JastAddJ5	28	0	100
JGC	28	0	100

4.1.1.3 If statement

กรณีทดสอบของ If statement ถูกจัดอยู่ภายใต้การทำงานของรูปประโยคเช่นเดียวกันกับ break statement และ continue statement โดยมีกรณีทดสอบทั้งสิ้น 23 กรณีโดยกรณีทดสอบนั้นเป็นดังตัวอย่างต่อไปนี้

```
public class T149is4 {
    T149is4 (){}
    public static void main(String[] args) {

        if (true)
            ;
            ;
        else
            ;
    }
}
```

ผลการทดสอบจากกรณีทดสอบทั้งหมดข้างต้นถูกแสดงอยู่ในตารางที่ 4.4 โดยมีผลเป็นดังนี้

ตารางที่ 4.4 แสดงผลการทดสอบของ If statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	23	0	100
JastAddJ5	23	0	100
JGC	23	0	100

4.1.1.4 Labeled statement

มีกรณีทดสอบ 22 กรณีสำหรับใช้ตรวจสอบไวยากรณ์ของ Labeled statement โดยตัวอย่างของกรณีทดสอบนี้ถูกแสดงดังต่อไปนี้

```
public class T147enclosed1 {
    T147enclosed1 (){}
    public static void main(String[] args) {

        test:

    }
}
```

ผลการทดสอบชุดกรณีทดสอบ Labeled statement ทั้งหมดถูกแสดงอยู่ในตารางที่ 4.5 โดยมีผลเป็นดังนี้

ตารางที่ 4.5 แสดงผลการทดสอบของ Labeled statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	22	0	100
JastAddJ5	22	0	100
JGC	22	0	100

4.1.1.5 Local class declarations

เพื่อประเมินการทำงานของตัวแปลภาษาในส่วนงานที่เกี่ยวข้องกับไวยากรณ์ของ Local class declarations จึงได้มีการใช้กรณีทดสอบ 12 กรณี โดยตัวอย่างของกรณีทดสอบข้างต้นสามารถแสดงออกมาในรูปแบบของภาษา Java ได้เป็นดังนี้

```
class T143s11 {
  void m(int i) {
      switch (i) {
        case 0:
        class Local {}
        break;
        case 1:
        class Local {}
      }
  }
}
```

และผลการทดสอบชุดกรณีทดสอบของ Local class declarations ทั้งหมดถูกแสดงอยู่ในตารางที่ 4.6 โดยมีผลเป็นดังนี้

ตารางที่ 4.6 แสดงผลการทดสอบของ Local class declarations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	12	0	100
JastAddJ5	12	0	100
JGC	12	0	100

4.1.1.6 Switch statement

กรณีทดสอบเพื่อตรวจสอบการทำงานของส่วนคัดแยกคำและส่วนของการตรวจสอบไวยากรณ์ที่เกี่ยวข้องกับ Switch statement มีทั้งหมด 41 กรณีซึ่งสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
class T1410e1 {
    void foo() {
        int i;
        switch (i = 1) {
            case 1:
        }
    }
}
```

จากผลการทดสอบจากกรณีทดสอบทั้งหมดข้างต้นถูกแสดงอยู่ในตารางที่ 4.7 โดยมีผลเป็นดังนี้

ตารางที่ 4.7 แสดงผลการทดสอบของ Switch statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	41	0	100
JastAddJ5	41	0	100
JGC	41	0	100

4.1.1.7 Local class declaration statements

กรณีทดสอบของ Local class declaration statements นั้นแตกต่างจาก Local class declaration โดยกรณีทดสอบนี้จะประกอบไปด้วยรูปไวยากรณ์ของประโยค โดยสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
public class T1442is1 {
    T1442is1 (){}
    public static void main(String[] args) {

        int i;
        int i;

    }
}
```

จากกรณีทดสอบทั้งหมด 24 กรณีผลการทดสอบถูกแสดงอยู่ในตารางที่ 4.8 โดยมีผลเป็นดังนี้

ตารางที่ 4.8 แสดงผลการทดสอบของ Local class declaration statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	24	0	100
JastAddJ5	24	0	100
JGC	24	0	100

4.1.1.8 Synchronized statement

มีการออกแบบกรณีทดสอบ 14 กรณีสำหรับใช้ตรวจสอบไวยากรณ์ของ Synchronized statement โดยตัวอย่างของกรณีทดสอบนี้ถูกแสดงเป็นภาษา Java ดังต่อไปนี้

```
class T141813 {

        static synchronized void foo() {
            synchronized (T141813.class) {}
        }
}
```

ซึ่งผลลัพธ์ของการทดสอบของกรณี Synchronized statement แสดงอยู่ในตารางที่ 4.9

ตารางที่ 4.9 แสดงผลการทดสอบของ Synchronized statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	14	0	100
JastAddJ5	14	0	100
JGC	14	0	100

4.1.1.9 Try statement

กรณีทดสอบ 64 กรณีถูกนำมาทดสอบการทำงานของตัวแปลภาษาในส่วนที่เกี่ยวข้องกับ Try statement โดยแต่ละกรณีทดสอบนั้นได้นำเงื่อนไขต่างๆ ที่คำสำคัญอย่าง `try` สามารถดำเนินการได้กรณีทดสอบข้างต้นบางส่วนสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
class T1419e1 {

  void m() {
      try {
        throw new Exception();
      } catch (RuntimeException e) {
      }
  }
}
```

ผลลัพธ์ของการทดสอบไวยากรณ์ของกรณี If statement แสดงอยู่ใน ตารางที่ 4.10

ตารางที่ 4.10 แสดงผลการทดสอบของ Try statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	64	0	100
JastAddJ5	64	0	100
JGC	64	0	100

4.1.1.10 Throw statement

กรณีทดสอบทั้งหมด 24 กรณีได้ถูกนำมาทดสอบการทำงานของตัวแปลภาษาทั้ง 3 ตัวได้แก่ Javac JastAddJ5 และ JGC ซึ่ง Throw statement นั้นเกี่ยวข้องกับส่วนต่างๆ ของไวยากรณ์ของภาษา Java ในหลายลักษณะโดยกรณีทดสอบบางส่วนสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
class t1417c1 {
            void m() {
            throw new RuntimeException(); // unchecked
        }
}
```

จากกรณีทดสอบทั้งหมด 24 กรณีผลการทดสอบถูกแสดงอยู่ในตารางที่ 4.11 โดยมีผลเป็นดังนี้

ตารางที่ 4.11 แสดงผลการทดสอบของ Throw statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	24	0	100
JastAddJ5	24	0	100
JGC	24	0	100

4.1.1.11 Unreachable statement

การตรวจสอบการทำงานของ Unreachable statement ถือเป็นหนึ่งในงานสำคัญของงานด้านตัวแปลภาษา ซึ่งจะช่วยตรวจสอบความถูกต้องของโปรแกรม ทำให้ลดการทำงานที่ไม่เหมาะสม และทำให้นักพัฒนาโปรแกรมสามารถควบคุมระบบได้ง่าย กรณีทดสอบ 224 กรณีถูกใช้มาเพื่อตรวจสอบการทำงานของตัวแปลภาษาด้วยกรณีทดสอบที่มีจำนวนมากเป็นข้อพิสูจน์ให้เห็นว่านี่เป็นหนึ่งในงานสำคัญของระบบ โดยกรณีทดสอบบางส่วนสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
public class T1420ful {
    T1420ful (){}
    public static void main(String[] args) {

        for (int i = 0; ; i++) break;

    }
}
```

ผลลัพธ์ของการทดสอบไวยากรณ์ของกรณี Unreachable statement แสดงอยู่ในตารางที่ 4.12

ตารางที่ 4.12 แสดงผลการทดสอบของ Unreachable statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	224	0	100
JastAddJ5	224	0	100
JGC	224	0	100

ในการทดสอบนี้ตัวแปลภาษาทั้ง 3 ตัวคือ Javac JastAddJ5 และ JGC ได้ถูกประเมินในส่วนไวยากรณ์ของช่องและรูปประโยคด้วยกรณีทดสอบ 503 กรณี โดยในส่วนถัดไปของการทดสอบจะเป็นการทดสอบไวยากรณ์ของคลาส

4.1.3 การทดสอบไวยากรณ์ของคลาส (An experiment of classes)

กระบวนการตรวจสอบนี้เป็นการตรวจสอบไวยากรณ์ต่างๆ โดยเฉพาะที่เกี่ยวข้องกับคลาส กระบวนการนี้จะประกอบไปด้วยส่วนย่อยหลายส่วนด้วยกัน อย่างเช่น class members และ constructor declaration เป็นต้น โดยส่วนย่อยต่างๆ จะถูกอธิบายการทดสอบในส่วนต่อไป

4.1.3.1 Class declaration

โครงสร้างไวยากรณ์ของคลาสนั้นจะประเมินด้วยโครงสร้างที่เรียบง่ายไปจนกระทั่งโครงสร้างที่มีความซับซ้อนมากขึ้น ตัวอย่างของกรณีทดสอบที่นำมาประเมินความสามารถของตัวแปลภาษานั้นถูกแสดงอยู่ด้านล่าง

```
class T81e2 {
  static class Foo {
      // even though there is no enclosing instance T81e2,
this clashes
      class T81e2 {}
  }
}
```

ผลลัพธ์ของการทดสอบไวยากรณ์ของกรณี Class declaration มีทั้งสิ้น 137 กรณีแสดงอยู่ในตารางที่ 4.13

ตารางที่ 4.13 แสดงผลการทดสอบของ Class declaration

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	137	0	100
JastAddJ5	137	0	100
JGC	137	0	100

4.1.3.2 Class member

มีกรณีทดสอบ 9 กรณีถูกเตรียมมาทดสอบไวยากรณ์ในงานด้าน Class member โดยกรณีทดสอบบางส่วนถูกแสดงในรูปแบบภาษา Java ได้เป็นดังนี้

```
class T82aim1 {
    public int foo;
}

class T82aim1_Test extends T82aim1 {
    void bar() { foo = 0; }
}
```

ผลลัพธ์ของการทดสอบไวยากรณ์ของกรณี Class member ทั้งหมดแสดงอยู่ในตารางที่ 4.14

ตารางที่ 4.14 แสดงผลการทดสอบของ Class member

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	9	0	100
JastAddJ5	9	0	100
JGC	9	0	100

4.1.3.3 Constructor declaration

ในส่วนการตรวจสอบไวยากรณ์ที่เกี่ยวข้องกับ Constructor declaration นั้นมีส่วนประกอบย่อยหลายส่วนอย่างเช่น constructor-body constructor-modifiers, constructor-overloading, constructor-signature, default-constructor และ formal-parameters โดยชุดกรณีทดสอบเหล่านี้ถูกออกแบบและทำงานภายใต้งานในลักษณะนี้ โดยตัวอย่างของกรณีทดสอบสามารถแสดงอยู่ในรูปแบบภาษา Java ได้เป็นดังนี้

```
class T8851aci1 {

 T8851aci1() { this(0); }
 T8851aci1(int i) {}

}
```

กรณีทดสอบดังกล่าวข้างต้นของส่วนนี้มีทั้งหมด 130 กรณีทดสอบโดยผลการทดสอบถูกแสดงอยู่ในตารางที่ 4.15

ตารางที่ 4.15 แสดงผลการทดสอบของ Constructor declaration

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	128	2	98.46
JastAddJ5	130	0	100
JGC	130	0	100

4.1.3.4 Field declarations

กรณีทดสอบ 176 กรณีถูกนำมาทดสอบการทำงานของตัวแปลภาษาในส่วนที่เกี่ยวข้องกับ Field declarations โดยตัวอย่างกรณีทดสอบข้างต้นบางส่วนสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
class T83h1 {

  class One {
      public int pub;
      protected int pro;
      int pack;
      private int priv;
  }
  class Two extends One {
      private int pub;
      private int pro;
      private int pack;
      private int priv;
  }
  class Three extends One {
      int pub;
      int pro;
      int pack;
      int priv;
  }
  class Four extends One {
      protected int pub;
      protected int pro;
      protected int pack;
      protected int priv;
  }
  class Five extends One {
      public int pub;
      public int pro;
      public int pack;
      public int priv;
  }
}
```

ผลลัพธ์ของการทดสอบทั้ง 176 กรณีถูกแสดงอยู่ในตารางที่ 4.16 ซึ่งกรณีทดสอบทั้งหมดถูกออกแบบมาจาก JLS

ตารางที่ 4.16 แสดงผลการทดสอบของ Field declarations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	176	0	100
JastAddJ5	176	0	100
JGC	176	0	100

4.1.3.5 Instance initializers

ไวยากรณ์สำหรับการกำหนดการสร้างตัวแปรที่เป็นลักษณะการทำงานของการเขียนโปรแกรมเชิงวัตถุเป็นหนึ่งในงานที่สำคัญสำหรับการเขียนโปรแกรมด้วยภาษา Java โดยใช้กรณีทดสอบทั้งหมด 34 กรณีมาใช้ตรวจสอบการทำงานของตัวแปลภาษา โดยตัวอย่างกรณีทดสอบข้างต้นบางส่วนสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
class T86ce6 {

        void foo() throws ClassNotFoundException {
         new Object() {
         { if (true) throw new ClassNotFoundException(); }
         };
        }
}
```

จากการทดสอบของชุดกรณีทดสอบข้างต้นผลการทดสอบการทำงานของตัวแปลภาษาทั้ง 3 ตัว คือ Javac JastAddJ5 และ JGC ถูกแสดงอยู่ในตารางที่ 4.17

ตารางที่ 4.17 แสดงผลการทดสอบของ Instance initializers

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	34	0	100
JastAddJ5	34	0	100
JGC	34	0	100

4.1.3.6 Member type declarations

มีกรณีทดสอบ 55 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษา โดยมีกรณีทดสอบบางกรณีจะเป็นการทดสอบของคลาสแบบภายใน (inner class) โดยอยู่ในรูปของสมาชิกของคลาส กรณีทดสอบเหล่านี้สามารถแสดงให้อยู่ในรูปแบบภาษา Java ได้ดังนี้

```
class T852smu5 {

        interface I {}
        static class T852smu5Test {
            I i;
        }
}
```

จากกรณีทดสอบทั้ง 55 กรณีผลการทดสอบการทำงานของตัวแปลภาษาทั้ง 3 ตัวถูกแสดงอยู่ในตารางที่ 4.18

ตารางที่ 4.18 แสดงผลการทดสอบของ Member type declarations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	54	1	98.18
JastAddJ5	55	0	100
JGC	55	0	100

4.1.3.7 Method declarations

กรณีทดสอบ 208 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาโดยอ้างอิงมาจาก JLS ชุดกรณีทดสอบชุดนี้ประกอบไปด้วยหน่วยย่อยหลายส่วนคือ inheritance-overriding-and-hiding, constructor-modifiers, method-modifiers, method-signature และ method-throws กรณีทดสอบเหล่านี้สามารถแสดงให้อยู่ในรูปแบบภาษา Java ได้ดังนี้

```
public class ConflictingReturnInNonVoid {
    public boolean foo() {
        int i = 0;
        if (i == 1) {
            return true;
        } else {
            return;
        }
    }
}
```

จากการทดสอบด้วยกรณีทดสอบทั้งหมด 208 กรณีผลการทดสอบถูกแสดงอยู่ในตารางที่ 4.19

ตารางที่ 4.19 แสดงผลการทดสอบของ Method declarations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	208	0	100
JastAddJ5	208	0	100
JGC	208	0	100

4.1.3.8 Static Initializers

กรณีทดสอบ 35 กรณีถูกนำมาทดสอบการทำงานของตัวแปลภาษาในส่วนที่เกี่ยวข้องกับ Static Initializers โดยแต่ละกรณีทดสอบนั้นได้นำเงื่อนไขต่างๆ ที่คำสำคัญอย่าง `static` โดยกรณีทดสอบข้างต้นบางส่วนสามารถแปลงออกมาเป็นรหัสต้นฉบับของภาษา Java ได้ดังนี้

```
class T87r3 {
        static { return 1; }
}
```

ผลการทดสอบจากกรณีทดสอบทั้งหมดถูกแสดงอยู่ในตารางที่ 4.20 โดยผลต่างๆ เป็นดังนี้

ตารางที่ 4.20 แสดงผลการทดสอบของ Static initializers

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	35	0	100
JastAddJ5	35	0	100
JGC	35	0	100

ในการทดสอบไวยากรณ์ของคลาสนี้ได้นำกรณีทดสอบจำนวน 784 กรณีมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิดได้แก่ Javac JastAddJ5 และ JGC โดยในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของการเปลี่ยนรูปและเพิ่มลักษณะตัวแปร

4.1.4 การทดสอบไวยากรณ์ของการเปลี่ยนรูปและเพิ่มลักษณะตัวแปร (An experiment of conversions and promotions)

ชุดทบสอบของการทดสอบไวยากรณ์ของการเปลี่ยนรูปและเพิ่มลักษณะตัวแปร ประกอบไปด้วย 2 ส่วนย่อยคือ assignment-conversion และ kinds-of-conversion โดยกรณีทดสอบสามารถแสดงออกมาอยู่ในรูปแบบของภาษา Java ได้ดังนี้

```
public class T52ae1 {
    T52ae1 (){}
    public static void main(String[] args) {

        Object[] o = 0.0D;

    }
}
```

ในการทดสอบไวยากรณ์ของไวยากรณ์ของการเปลี่ยนรูปและเพิ่มลักษณะตัวแปรได้นำกรณีทดสอบจำนวน 255 กรณีมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิดคือ Javac JastAddJ5 และ JGC โดยตารางที่ 4.21 ได้แสดงผลการทดสอบเอาไว้ และในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของไวยากรณ์ของการกำหนดค่า

ตารางที่ 4.21 แสดงผลการทดสอบของ Conversions and promotions

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	254	1	99.60
JastAddJ5	255	0	100
JGC	255	0	100

4.1.5 การทดสอบไวยากรณ์การกำหนดค่าต่างๆ (An experiment of Definite assignment)

การทดสอบไวยากรณ์การกำหนดค่าต่างๆ แน่นอนว่าจะต้องประกอบไปด้วยกรณีทดสอบย่อยต่างๆ อาทิเช่น anonymous-classes, constructors-and-instance-initializers, expressions และ statements โดยขั้นตอนพื้นฐานของการทดสอบนี้จะเริ่มด้วยกรณีทดสอบอย่างง่าย 10 กรณีโดยสามารถแปลงออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
class T16i1 {

        final int i;
        {
            new T16i1().i = 1;
        }
}
```

ผลการทดสอบไวยากรณ์ของไวยากรณ์พื้นฐานของการกำหนดค่าถูกแสดงเอาไว้ใน ตารางที่ 4.22

ตารางที่ 4.22 แสดงผลการทดสอบของ Definite assignment

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	10	0	100
JastAddJ5	10	0	100
JGC	10	0	100

4.1.5.1 Anonymous class

เพื่อประเมินการทำงานในส่วนของ Anonymous class ดังนั้นมีการเตรียมกรณีทดสอบที่ได้จาก JLS ซึ่งมี 3 กรณีทดสอบโดยสามารถแปลงออกมาเป็นรหัสต้นฉบับภาษา Java ได้ดังนี้

```
class T87r3 {

    static { return 1; }

}
```

ผลการทดสอบไวยากรณ์ของไวยากรณ์พื้นฐานของ Anonymous class ถูกแสดงเอาไว้ในตารางที่ 4.23

ตารางที่ 4.23 แสดงผลการทดสอบของ Anonymous class

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	3	0	100
JastAddJ5	3	0	100
JGC	3	0	100

4.1.5.2 Constructors and instance initializers

มีการออกแบบกรณีทดสอบ 12 กรณีสำหรับใช้ประเมินการทำงานของตัวแปลภาษาในส่วนของ constructors-and-instance-initializers ซึ่งการทดสอบในส่วนนี้ใช้ประเมินการตั้งค่าในส่วนของ constructors กรณีโดยสามารถแปลงออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
class T168p3 {

  class Super {}
  class Sub extends Super {
      Sub(final boolean b) {
        ((b = true) ? new T168p3() : null).super();
      }
  }
}
```

ผลการทดสอบของส่วน constructors-and-instance-initializers แสดงอยู่ในตารางที่ 4.24

ตารางที่ 4.24 แสดงผลการทดสอบของ Constructor and instance initializers

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	12	0	100
JastAddJ5	12	0	100
JGC	12	0	100

4.1.5.3 Expressions

มีชุดทดสอบย่อย 4 ชุดสำหรับประเมินการทำงานของตัวแปลภาษาในส่วนของนิพจน์ ได้แก่ assignment-expressions, boolean-conditional-and, boolean-conditional-operator, boolean-conditional-or และ operators-increment-and-decrement รวมกรณีทดสอบทั้งหมด 241 กรณีโดยสามารถแปลงออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
public class T1618daf7 {
    T1618daf7 (){}
    public static void main(String[] args) {

 int i;
 ++(new int[] {0})[i];
       }
}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาโดยกรณีทดสอบทั้งหมด 241 กรณีแสดงอยู่ในตารางที่ 4.25

ตารางที่ 4.25 แสดงผลการทดสอบของ Expression

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	237	4	98.34
JastAddJ5	237	4	98.34
JGC	237	4	98.34

4.1.5.4 Statements

การทดสอบนี้เป็นหนึ่งในการทดสอบที่สำคัญสำหรับไวยากรณ์ของภาษา Java ที่บรรจุอยู่ในภาษา JGroovy โดยใช้ในการประเมินการทำงานของตัวแปลภาษาในส่วนของรูปประโยคต่างๆ โดยการทดสอบนี้จะประกอบไปด้วยส่วนย่อยหลายส่วน ได้แก่ do-statements, for-statements, if-statements, labeled-statements, local-class-declaration-statements, switch-statement, try-statements และ while-statements ซึ่งมีกรณีทดสอบทั้งหมด 160 กรณีโดยตัวอย่างที่อยู่ในรูปของภาษา Java เป็นดังนี้

```
public class T16210daf1 {
    T16210daf1 (){}
    public static void main(String[] args) {

 int i;
 do
    break;
 while (true);
 int j = i;

    }
}
```

ผลจากการทดสอบการทำงานของกรณีทดสอบทั้ง 160 กรณีแสดงอยู่ในตารางที่ 4.26

ตารางที่ 4.26 แสดงผลการทดสอบของ Statements

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	160	0	100
JastAddJ5	160	0	100
JGC	160	0	100

ในการทดสอบไวยากรณ์การกำหนดค่าต่างๆ นี้ได้นำกรณีทดสอบจำนวน 416 กรณี มาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิดคือ Javac JastAddJ5 และ JGC โดยในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของนิพจน์

4.1.6 การทดสอบไวยากรณ์ของนิพจน์ (An experiment of expressions)

การทดสอบในส่วนนี้เป็นส่วนทดสอบหลักของไวยากรณ์ที่เป็นนิพจน์ โดยจะประกอบไปด้วยการทดสอบนิพจน์ในส่วนต่างๆ ของโครงสร้างภาษาทำให้มีหน่วยทดสอบย่อยหลายหน่วยได้แก่ additive-operators, array-access-expressions, array-creation-expressions, assignment-operators, cast-expressions, class-instance-creation, conditional-and-operator, conditional-operator, conditional-or-operator, constant-expression, equality-operators, field-access-expressions, method-invocation-expressions, multiplicative-operators, normal-and-abrupt-completion, postfix-expressions, primary-expressions, relational-operators และ unary-operators โดยในส่วนต่อไปจะเป็นการอธิบายหน่วยทดสอบย่อยแต่ละหน่วย

4.1.6.1 Additive operator

การทดสอบในส่วนย่อยนี้เพื่อประเมินการทำงานของตัวแปลภาษาต่อเครื่องหมายบวก (+) โดยจะทดสอบตัวแปลภาษาด้วยกรณีทดสอบ 110 กรณีดังตัวอย่างต่อไปนี้

```
public class T15181nc8 {
    T15181nc8 (){}
     public static void main(String[] args) {

         int i = 10;
         String s2 = "" + i;
          s2 = i + "";

     }
}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาทั้ง 110 กรณีแสดงอยู่ในตารางที่ 4.27

ตารางที่ 4.27 แสดงผลการทดสอบของ Additive operator

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	102	8	92.72
JastAddJ5	102	8	92.72
JGC	102	8	92.72

4.1.6.2 Array access expressions and array creation expressions

มีกรณีทดสอบ 55 กรณีตามมาตรฐานของ JLS ได้ถูกนำมาประเมินความสามารถการทำงานของตัวแปลภาษาโดยชุดกรณีทดสอบนี้อยู่ในส่วนทดสอบย่อยของ array-access-expressions sub-package และ array-creation-expressions โดยตัวอย่างของกรณีทดสอบจะเป็นดังนี้

```
public class T1513t3 {
    T1513t3 (){}
    public static void main(String[] args) {

 final Object[] s = {""};
 s[0].valueOf(1);

    }
}
```

ผลจากการทดสอบการทำงานของกรณีทดสอบทั้ง 55 กรณีของส่วนทดสอบย่อย array-access-expressions sub-package และ array-creation-expressions แสดงอยู่ในตารางที่ 4.28

ตารางที่ 4.28 แสดงผลการทดสอบของ Array access expressions and array creation expressions

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	102	8	92.72
JastAddJ5	102	8	92.72
JGC	102	8	92.72

4.1.6.3 Assignment operator and cast expression

การทดสอบย่อยนี้เป็นการรวมชุดทดสอบของกรณี Assignment operator และ cast expression เข้ามารวมกันอยู่ในหน่วยเดียวโดยมีกรณีทดสอบทั้งหมด 268 กรณี โดยกรณีทดสอบตัวอย่างที่อยู่ในรูปของภาษา Java เป็นดังนี้

```
public class T1516r2 {
    T1516r2 (){}
    public static void main(String[] args) {

  String o = null;
  String s = (String) o;

    }
}
```

ผลจากการทดสอบการทำงานของกรณีทดสอบทั้ง 268 กรณีของส่วนทดสอบย่อย Assignment operator และ cast expression แสดงอยู่ในตารางที่ 4.29

ตารางที่ 4.29 แสดงผลการทดสอบของ Assignment operator and cast expression

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	265	3	98.88
JastAddJ5	268	0	100
JGC	268	0	100

4.1.6.4 Class instance creation

กระบวนการสร้างวัตถุเป็นกระบวนการที่สำคัญในการเขียนโปรแกรมเชิงวัตถุ โดยเป็นหนึ่งในไวยากรณ์ที่ต้องได้รับการตรวจสอบเป็นอย่างดีซึ่งจะใช้กรณีทดสอบ 127 กรณีเพื่อประเมินตัวแปลภาษาในลักษณะงานดังกล่าว โดยกรณีทดสอบตัวอย่างที่อยู่ในรูปของภาษา Java เป็นดังนี้

```
package p2;
import p1.*;
class T1591ua13_2 implements p1.T1591ua13_1 {
    Object o1 = new p1.T1591ua13_1(){};
    Object o2 = new T1591ua13_1(){};
}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาทั้ง 127 กรณีแสดงอยู่ในตารางที่ 4.30

ตารางที่ 4.30 แสดงผลการทดสอบของ Class instance creation

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	127	0	100
JastAddJ5	127	0	100
JGC	127	0	100

4.1.6.5 Conditional operator

มีส่วนทดสอบย่อยหลายส่วนที่เป็นชุดกรณีทดสอบซึ่งทำงานภายใต้ Conditional operator ได้แก่ conditional-and-operator และ conditional-or-operator โดยมีกรณีทดสอบ 83 กรณีโดยกรณีทดสอบตัวอย่างที่อยู่ในรูปของภาษา Java เป็นดังนี้

```
public class T1524o2 {
    T1524o2 (){}
    public static void main(String[] args) {

      boolean b = true;
       if (true || b);
       if (true || !b);
       b = false;

    }
}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาทั้ง 83 กรณีแสดงอยู่ในตารางที่ 4.31

ตารางที่ 4.31 แสดงผลการทดสอบของ Conditional operator

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	83	0	100
JastAddJ5	83	0	100
JGC	83	0	100

4.1.6.6 Constant expression

กรณีทดสอบ 109 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ตัวได้แก่Javac JastAddJ5 และ JGC โดยไวยากรณ์ของนิพจน์ค่าคงที่ถูกนำไปใช้ในหลายส่วนของการเขียนโปรแกรมซึ่งทำให้ต้องมีการทดสอบอย่างละเอียดโดยกรณีทดสอบตัวอย่างที่อยู่ในรูปของภาษา Java เป็นดังนี้

```
class T1528csn2 {

        final boolean t = true;
       void foo (int j) {
            switch (j) {
                case 0:
                case ((boolean) t ? 1 : 0):
            }
        }

}
```

ผลการทดสอบจากกรณีทดสอบ 109 กรณีที่ประเมินการทำงานในส่วนของนิพจน์ค่าคงที่นี้แสดงอยู่ในตารางที่ 4.32

ตารางที่ 4.32 แสดงผลการทดสอบของ Constance expression

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	109	0	100
JastAddJ5	109	0	100
JGC	109	0	100

4.1.6.7 Equality operator, relational operator and unary operator

ในการประเมินการทำงานของตัวแปลภาษาในส่วนนี้ประกอบไปด้วยหน่วยทดสอบย่อย 3 ส่วนคือ equality-operator, relational-operator และ the unary-operator โดยมีกรณีทดสอบทั้งหมด 238 กรณีซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
public class T15151r2 {
    T15151r2 (){}
    public static void main(String[] args) {

        int i = 1;
        ++(++i);

    }
}
```

ผลการทดสอบทั้ง 238 กรณีแสดงอยู่ในตารางที่ 4.33 ซึ่งผลการทดสอบเป็นดังต่อไปนี้

ตารางที่ 4.33 แสดงผลการทดสอบของ Equality operator, relational operator and unary operator

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	232	6	100
JastAddJ5	236	2	100
JGC	236	2	100

4.1.6.8 Field access expression and method invocation expression

ด้วยกรณีทดสอบ 155 กรณีที่อยู่ภายใต้การทดสอบของ 2 หน่วยทดสอบย่อย field-access-expressions และ method-invocation-expressions ได้ถูกนำมาประเมินประสิทธิภาพการทำงานของตัวแปลภาษาซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
class T15123a1 {

 abstract class A {
     abstract void m();
 }
 abstract class C extends A {
     { super.m(); }
 }
}
```

ผลจากการประเมินของ 2 หน่วยทดสอบย่อย field-access-expressions และ method-invocation-expressions แสดงอยู่ในตารางที่ 4.34

ตารางที่ 4.34 แสดงผลการทดสอบของ Field access expression and method invocation expression

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	155	0	100
JastAddJ5	155	0	100
JGC	155	0	100

4.1.6.9 Multiplicative operators

การทดสอบในส่วนย่อยนี้เพื่อประเมินการทำงานของตัวแปลภาษาต่อเครื่องหมายคูณ (x) โดยจะทดสอบตัวแปลภาษาด้วยกรณีทดสอบ 159 กรณีดังตัวอย่างต่อไปนี้

```
class T15171f9 {
    T15171f9 (){}
    void foo(int i) {
        switch (i) {
            case 0:
            case ((1.0e30f * 1.0e30f ==
Float.POSITIVE_INFINITY) ? 1 : 0):
            case ((1.0e30f * -1.0e30f ==
Float.NEGATIVE_INFINITY) ? 2 : 0):
        }
    }
}
```

ผลการทดสอบทั้ง 159 กรณีแสดงอยู่ในตารางที่ 4.35 ซึ่งผลการทดสอบเป็นดังต่อไปนี้

ตารางที่ 4.35 แสดงผลการทดสอบของ Multiplicative operators

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	159	0	100
JastAddJ5	159	0	100
JGC	159	0	100

4.1.6.10 Postfix expressions and primary expression

ด้วยกรณีทดสอบ 154 กรณีที่อยู่ภายใต้การทดสอบของ 2 หน่วยทดสอบย่อย postfix-expressions และ primary expressions ได้ถูกนำมาประเมินประสิทธิภาพการทำงานของตัวแปลภาษาซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
public class T15142r2 {
    T15142r2 (){}
    public static void main(String[] args) {

        int i = 1;
        (i--)--;

    }
}
```

ผลจากการประเมินของ 2 หน่วยทดสอบย่อย postfix-expressions และ primary expressions แสดงอยู่ในตารางที่ 4.36

ตารางที่ 4.36 แสดงผลการทดสอบของ Postfix expressions and primary expression

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	151	3	98.05
JastAddJ5	151	3	98.05
JGC	151	3	98.05

ในการทดสอบไวยากรณ์ของนิพจน์นี้ได้นำกรณีทดสอบจำนวน 1455 กรณีมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิดได้แก่ Javac JastAddJ5 และ JGC โดยในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของอินเตอร์เฟส

4.1.7 การทดสอบไวยากรณ์ของอินเตอร์เฟส (An experiment of interfaces)

ในส่วนการทดสอบไวยากรณ์ของอินเตอร์เฟสประกอบไปด้วยส่วนทดสอบย่อยหลายส่วนด้วยกันคือ abstract-method-declarations, field-constant-declarations, interface-declarations และ interface-members โดยส่วนทดสอบที่สำคัญต่างๆ นั้นถูกอธิบายในหัวข้อย่อยต่อไปนี้

4.1.7.1 Abstract method declarations

การทดสอบในส่วนของ Abstract method declarations สามารถแบ่งออกเป็นชุดข้อมูล 2 ชุดคือ inheritance-and-overriding และ overloading โดยจะประกอบไปด้วยกรณีทดสอบ 28 กรณีซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
class T94c3 {

        interface I1 { void m(); }
        interface I2 { void m(); }
        interface I3 extends I1, I2 {}
        void foo(I3 i) {
            i.m();
        }

}
```

ผลจากการประเมินของ 2 หน่วยทดสอบย่อย inheritance-and-overriding และ overloading โดยแสดงอยู่ในตารางที่ 4.37

ตารางที่ 4.37 แสดงผลการทดสอบของ Abstract method declarations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	28	0	100
JastAddJ5	28	0	100
JGC	28	0	100

4.1.7.2 Field constant declaration

ในการทดสอบส่วนนี้จะประกอบไปด้วยการทดสอบค่าคงที่และการกำหนดค่าเริ่มต้นโดยมีกรณีทดสอบ 33 กรณีสำหรับการตรวจสอบความถูกต้องของตัวแปลภาษา ซึ่งสามารถแปลงเป็นภาษา Java ได้ดังนี้

```
class T93c4 {

        interface I1 { int i = 1; }
        interface I2 extends I1 {}
        interface I3 extends I1, I2 { int j = i; }

}
```

จากผลการทดสอบทั้ง 33 กรณีของการทดสอบส่วน field-constant-declarations แสดงอยู่ในตารางที่ 4.38

ตารางที่ 4.38 แสดงผลการทดสอบของ Field constant declaration

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	33	0	100
JastAddJ5	33	0	100
JGC	33	0	100

4.1.7.3 Interface declaration

มีกรณีทดสอบ 98 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ตัวในส่วนของการประกาศอินเตอร์เฟสซึ่งครอบคลุม 3 ส่วนย่อยของชุดกรณีทดสอบได้แก่ interface-body-and-member-declarations, interfacemodifiers และ superinter-faces-and-subinterfaces โดยตัวอย่างของกรณีทดสอบเป็นดังนี้

```
class T912a1a {
    private interface I {}
}
interface T912a1b extends T912a1a.I {}
```

จากผลการทดสอบทั้ง 33 กรณีของส่วนการทดสอบตัวแปลภาษาทั้ง 3 ชนิดได้แก่Javac JastAddJ5 และ JGC แสดงอยู่ในตารางที่ 4.39

ตารางที่ 4.39 แสดงผลการทดสอบของ Interface declaration

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	98	0	100
JastAddJ5	98	0	100
JGC	98	0	100

4.1.7.4 Interface members

ในส่วนการทดสอบของ interface-members นี้จะประกอบไปด้วยกรณีทดสอบ 19 กรณีสำหรับประเมินการทำงานของตัวแปลภาษาโดยตัวอย่างของกรณีทดสอบเป็นดังนี้

```
interface T92i6 extends Cloneable {
    class Inner {
        Object bar(T92i6 i) {
            try {
                // Because this call is nested in the
interface, it would have
                // full access to protected i.clone() if
that existed.
                return i.clone();
            } catch (CloneNotSupportedException e) {
                return null;
            }
        }
    }
}
```

จากผลการทดสอบทั้ง 19 กรณีของส่วนการทดสอบ interface-members แสดงอยู่ในตารางที่ 4.40

ตารางที่ 4.40 แสดงผลการทดสอบของ Interface members

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	19	0	100
JastAddJ5	19	0	100
JGC	19	0	100

ในการทดสอบไวยากรณ์ของอินเตอร์เฟสนี้ประกอบไปด้วยกรณีทดสอบจำนวน 416 กรณีซึ่งนำถูกนำมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิดได้แก่ Javac JastAddJ5 และ JGC และในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของคำต่างๆ

4.1.8 การทดสอบโครงสร้างคำต่างๆ (An experiment of lexical structure)

การตรวจสอบโครงสร้างของคำต่างๆ เป็นหน้าที่สำคัญที่ละเลยไม่ได้ โดยกระบวนการทดสอบนี้เป็นหนึ่งในการทดสอบหลักซึ่งจะประกอบไปด้วยหน่วยการทดสอบย่อยต่างๆ ได้แก่ comments, identifiers, input-elements-and-token, keywords, lexical-transitions, line-terminators, literals, string-literals, unicode, unicode-escape และ white-space

4.1.8.1 Identifiers and keywords

ส่วนการทดสอบของตัวระบุชื่อและคำสำคัญเป็นการระบุความหมายของคำซึ่งทั้งคู่ถูกจัดอยู่ในส่วนเดียวกันซึ่งทำงานภายใต้ตัวคัดแยกคำสถานะจำกัดแบบลำดับชั้น โดยการทดสอบนี้มีกรณีทดสอบทั้งหมด 99 กรณีโดยตัวอย่างของกรณีทดสอบเป็นดังนี้

```
class T3913 {int double;}
```

จากผลการทดสอบทั้ง 99 กรณีของส่วนการทดสอบตัวระบุชื่อและคำสำคัญถูกแสดงอยู่ในตารางที่ 4.41

ตารางที่ 4.41 แสดงผลการทดสอบของ Identifier and keywords

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	99	0	100
JastAddJ5	99	0	100
JGC	99	0	100

4.1.8.2 Comment, line terminator and white space

ส่วนการทดสอบทั้ง 3 ส่วนนี้สามารถถูกจัดให้อยู่ในหมวดเดียวกัน โดยประกอบไปด้วยกรณีทดสอบทั้งหมด 140 กรณีซึ่งกรณีทดสอบตัวอย่างของส่วนนี้เป็นดังนี้

```
class T36v2 {int\u000ai;}

class T37ln3 { /*
 */ }oops
```

ผลการทดสอบจากกรณีทดสอบทั้ง 140 กรณีถูกแสดงอยู่ในตารางที่ 4.42

ตารางที่ 4.42 แสดงผลการทดสอบของ Comment, line terminator and white space

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	128	12	91.42
JastAddJ5	128	12	91.42
JGC	128	12	91.42

4.1.8.3 Input element and tokens

ส่วนการทดสอบนี้จะคล้ายคลึงกับการทดสอบในส่วนของตัวระบุชื่อและคำสำคัญโดยจะประกอบไปด้วยกรณีทดสอบทั้งหมด 17 กรณีด้วยกันซึ่งกรณีทดสอบตัวอย่างของส่วนนี้เป็นดังนี้

```
class T3512 {}
```

ผลการทดสอบกรณีทดสอบข้างต้นทั้งหมด 17 กรณีถูกบันทึกอยู่ในตารางที่ 4.43 โดยมีลักษณะข้อมูลดังนี้

ตารางที่ 4.43 แสดงผลการทดสอบของ Input element and tokens

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	17	0	100
JastAddJ5	17	0	100
JGC	17	0	100

4.1.8.4 Lexical translations

ในส่วนของการตรวจสอบการแปลงคำนี้มีกรณีทดสอบทั้งหมด 28 กรณีโดยสามารถแปลงออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
public class T32i11 {
    T32i11 (){}
    public static void main(String[] args) {

        int i = 0;
        i = i > > 1;

    }
}
```

ผลการทดสอบจากกรณีทดสอบในส่วนนี้ถูกบันทึกอยู่ในตารางที่ 4.44 โดยมีลักษณะข้อมูลดังนี้

ตารางที่ 4.44 แสดงผลการทดสอบของ Lexical translations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	28	0	100
JastAddJ5	28	0	100
JGC	28	0	100

4.1.8.5 Literal

ส่วนนี้คือส่วนที่เป็นข้อมูลของการเขียนโปรแกรมซึ่งเป็นถูกใช้ส่งข้อมูลให้ตัวแปรต่างๆ เพื่อตรวจสอบความถูกต้องของการทำงานการทดสอบจำเป็นต้องรัดกุม โดยมีกรณีทดสอบทั้งหมด 173 กรณีซึ่งสามารถแปลงออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
class T3101i8 {long l = 0x123456789abcdef01L;}
```

ผลการทดสอบจากกรณีทดสอบในส่วนนี้ถูกบันทึกอยู่ในตารางที่ 4.45 โดยมีลักษณะข้อมูลดังนี้

ตารางที่ 4.45 แสดงผลการทดสอบของ Literal

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	171	2	98.84
JastAddJ5	173	0	100
JGC	173	0	100

4.1.8.6 Unicode-escape

กรณีทดสอบ 13 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาในส่วนของ Unicode ซึ่งสามารถแปลงออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
class T33i3 {}/* \\u002a/
```

ผลการทดสอบจากกรณีทดสอบในส่วน Unicode ถูกบันทึกอยู่ในตารางที่ 4.46 โดยมีลักษณะข้อมูลดังนี้

ตารางที่ 4.46 แสดงผลการทดสอบของ Unicode-escape

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	13	0	100
JastAddJ5	13	0	100
JGC	13	0	100

ในการทดสอบนี้ตัวแปลภาษาทั้ง 3 ตัวคือ Javac JastAddJ5 และ JGC ได้ถูกประเมินในส่วนโครงสร้างคำต่างๆ ด้วยกรณีทดสอบ 470 กรณี โดยในส่วนถัดไปของการทดสอบจะเป็นการทดสอบชื่อ

4.1.9 การทดสอบชื่อ (An experiment of names)

การทดสอบในส่วนนี้จะทำการทดสอบของลำดับชั้นในการเรียกใช้ตัวแปร หรือเมธอดที่มีชื่อเดียวกันเพื่อประเมินการตรวจสอบความถูกต้องของการทำงานของตัวแปลภาษา โดยจะประกอบไปด้วย 5 ส่วนย่อยได้แก่ access-control, meaning-of-a-name, members-and-inheritance, names-and-identifiers, scope-of-a-declaration

4.1.9.1 Access control

การทดสอบในส่วนย่อยนี้จะประกอบไปด้วยกรณีทดสอบจำนวน 101 กรณี ซึ่งจากหน่วยทดสอบย่อย 2 หน่วยคือ details-on-protected-access และ determining-accessibility ซึ่งสามารถแปลงตัวอย่างของกรณีทดสอบออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
class T661a7 {
    Array7.A[] ref;
}
class Array7 {
    static class A {}
}
```

จากผลการทดสอบทั้ง 101 กรณีของการทดสอบส่วนการเข้าถึงแสดงอยู่ในตารางที่ 4.47

ตารางที่ 4.47 แสดงผลการทดสอบของ Access control

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	101	0	100
JastAddJ5	101	0	100
JGC	101	0	100

4.1.9.2 Meaning of a name

ส่วนการทดสอบความหมายของชื่อนี้เกี่ยวพันกับการทำงานของตัวแปรและเมธอด ซึ่งจะถูกใช้ในการทดสอบลำดับชั้นในการเข้าถึงของคลาส เมธอด และตัวแปรต่างๆ โดยจะประกอบไปด้วยส่วนย่อย 7 ส่วนคือ expression-names, method-names, package-names, packageortypenames, reclassification, syntactic-classification และ type-names โดยมีกรณีทดสอบ 158 กรณีซึ่งสามารถแปลงตัวอย่างของกรณีทดสอบออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
package p1;
public class T6551n14a {
    public interface T6551n14b {}
}
```

ผลการทดสอบจากกรณีทดสอบในส่วนความหมายของชื่อนี้ถูกบันทึกอยู่ในตารางที่ 4.48 โดยมีลักษณะข้อมูลดังนี้

ตารางที่ 4.48 แสดงผลการทดสอบของ Meaning of name

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	101	0	100
JastAddJ5	101	0	100
JGC	101	0	100

4.1.9.3 Member and inheritance

ลักษณะการตรวจสอบในส่วนนี้คล้ายกันกับความหมายของชื่อเพราะเกี่ยวกับการทำงานของตัวแปร เมธอด และคลาสซึ่งประกอบไปด้วยส่วนย่อย 4 ส่วน members-of-a-class-type, members-of-an-array-type, members-of-an-interface-type, members-of-a-package รวมกรณีทดสอบทั้งหมด 8 กรณีซึ่งสามารถแปลงตัวอย่างของกรณีทดสอบออกมาเป็นภาษา Java ได้ดังต่อไปนี้

```
interface One {
  int VAL = 1;
}

interface Two {
  int VAL = 2;
}

class QualifiedInterfaceMember implements One, Two {
  int i = One.VAL;
}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาโดยกรณีทดสอบทั้งหมด 8 กรณีแสดงอยู่ในตารางที่ 4.49

ตารางที่ 4.49 แสดงผลการทดสอบของ Member and inheritance

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	8	0	100
JastAddJ5	8	0	100
JGC	8	0	100

4.1.9.4 Name, identifiers and scope of a declaration

ส่วนการทดสอบนี้จะประกอบไปด้วย 2 ส่วนย่อยคือ names-and-identifiers และ scope-of-a-declaration โดยมีกรณีทดสอบ 17 กรณีโดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Java ได้ดังนี้

```
public class UseOfTypeBeforeDeclaration {
  Object o = new UseOfTypeBeforeDeclaration_Type();
}

class UseOfTypeBeforeDeclaration_Type {
  Object o = new UseOfTypeBeforeDeclaration();
}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาโดยกรณีทดสอบทั้งหมด 17 กรณีแสดงอยู่ในตารางที่ 4.50

ตารางที่ 4.50 แสดงผลการทดสอบของ Name, identifiers and scope of a declaration

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	17	0	100
JastAddJ5	17	0	100
JGC	17	0	100

ในการทดสอบชื่อนี้ประกอบไปด้วยกรณีทดสอบจำนวน 284 กรณีซึ่งนำถูกนำมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิด และในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของโครงสร้างหีบห่อ

4.1.10 การทดสอบโครงสร้างหีบห่อ (An experiment of package)

การประเมินโครงสร้างหีบห่อนี้ทำเพื่อตรวจสอบการเข้าถึงโครงสร้างในระดับต่างๆ จากส่วนที่แตกต่างกัน โดยมีส่วนทดสอบย่อย 5 ส่วนคือ compilation-units, import-declarations, package-declarations, package-members และ top-level-type-declarations โดยส่วนทดสอบที่สำคัญต่างๆ นั้นถูกอธิบายในหัวข้อย่อยต่อไปนี้

4.1.10.1 Compilation unit

มีกรณีทดสอบ 13 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาในส่วนการแปลนี้โดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Java ได้ดังนี้

```
package test; class T7313 {
    Object o;
    String s;
    Integer i;
}
```

ผลการทดสอบจากกรณีทดสอบข้างต้นทั้งหมด 13 กรณีแสดงอยู่ในตารางที่ 4.51

ตารางที่ 4.51 แสดงผลการทดสอบของ Compilation unit

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	13	0	100
JastAddJ5	13	0	100
JGC	13	0	100

4.1.10.2 Import declarations

การทดสอบในส่วนของ Import declarations สามารถแบ่งออกเป็นชุดข้อมูล 4 ชุดคือ automatic-imports, single-type-import-declaration, strange-example และ type-import-on-demand-declaration โดยจะประกอบไปด้วยกรณีทดสอบ 38 กรณีซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
import p1.T751d1a;
import p1.T751d1a;
class T751d1b extends T751d1a {}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาด้วยชุดข้อมูล 4 ชุด แสดงอยู่ในตารางที่ 4.52

ตารางที่ 4.52 แสดงผลการทดสอบของ Import declarations

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	38	0	100
JastAddJ5	38	0	100
JGC	38	0	100

4.1.10.3 Package declarations and package member

การทดสอบในส่วนนี้สามารถแบ่งออกเป็นชุดข้อมูล 2 ชุดคือ package-declarations และ package-members โดยจะประกอบไปด้วยกรณีทดสอบ 18 กรณีซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
package p1.T71n5;
class T71n5a {}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาด้วยชุดข้อมูล 2 ชุด แสดงอยู่ในตารางที่ 4.53

ตารางที่ 4.53 แสดงผลการทดสอบของ Package declarations and package member

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	17	1	94.44
JastAddJ5	18	0	100
JGC	18	0	100

4.1.10.4 Top level type declaration

การทดสอบในส่วนของ top-level-type-declaration ประกอบไปด้วยกรณีทดสอบ 17 กรณีซึ่งตัวอย่างของการทดสอบเป็นดังนี้

```
interface T76name2 extends T76name2pkg.pinterface {}
```

ผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนของ top-level-type-declaration แสดงอยู่ในตารางที่ 4.54

ตารางที่ 4.54 แสดงผลการทดสอบของ Top level type declaration

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	16	1	94.11
JastAddJ5	17	0	100
JGC	17	0	100

ในการทดสอบโครงสร้างหีบห่อนี้ประกอบไปด้วยกรณีทดสอบจำนวน 86 กรณีซึ่งนำถูกนำมาประเมินการทำงานของตัวแปลภาษาทั้ง 3 ชนิด และในหัวข้อต่อไปจะเป็นการทดสอบในส่วนของตัวแปรและชนิดของตัวแปร

4.1.11 การทดสอบตัวแปรและชนิดของตัวแปร(An experiment of Type value and variable)

มีกรณีทดสอบ 26 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาในส่วนตัวแปรและชนิดของตัวแปรนี้โดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Java ได้ดังนี้

```
class T454s4 {

        static final int val = 0;
        static final String str = null;
        static {
            val = 1;
            str = "";
        }

}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนตัวแปรและชนิดของตัวแปรแสดงอยู่ในตารางที่ 4.55

ตารางที่ 4.55 แสดงผลการทดสอบของ Type value and variable

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	26	0	100
JastAddJ5	26	0	100
JGC	26	0	100

4.1.12 การทดสอบ JVM (An experiment of JVM)

มีกรณีทดสอบ 91 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาใน JVM โดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Java ได้ดังนี้

```
class T4710jc5b extends T4710jc5a {}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนของ JVM แสดงอยู่ในตารางที่ 4.56

ตารางที่ 4.56 แสดงผลการทดสอบของ JVM

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	73	18	80.21
JastAddJ5	73	18	80.21
JGC	73	18	80.21

4.1.13 การทดสอบ Non JLS (An experiment of Non JLS)

กรณีทดสอบ 35 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาในส่วนของ non-jls โดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Java ได้ดังนี้

```
class Tnjrc2A { { Tnjrc2B.c.m(); } }
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนของ Non JLS แสดงอยู่ในตารางที่ 4.57

ตารางที่ 4.57 แสดงผลการทดสอบของ Non JLS

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	30	5	85.71
JastAddJ5	30	5	85.71
JGC	30	5	85.71

4.1.14 การทดสอบ Runtime (An experiment of runtime)

กรณีทดสอบ 4 กรณีถูกนำมาประเมินการทำงานของตัวแปลภาษาในส่วนของ Runtime โดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Java ได้ดังนี้

```
class T131c1 {
    int m(T131c1 t) { return t.i; }
    int j = m(this);
    final int i = 1;
    public static void main(String[] args) {
System.out.print(new T131c1().j);
    }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนของ Runtime แสดงอยู่ในตารางที่ 4.58

ตารางที่ 4.58 แสดงผลการทดสอบของ Runtime

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	4	0	100
JastAddJ5	4	0	100
JGC	4	0	100

ในการทดสอบนี้เป็นการประเมินการทำงานของตัวแปลภาษาส่วนหน้าของภาษา Java ด้วยกรณีทดสอบทั้งสิ้น 4619 กรณีโดยผลการทดสอบทั้งหมดแสดงอยู่ในตารางที่ 4.59

ตารางที่ 4.59 แสดงผลการทดสอบของตัวแปลภาษาส่วนหน้าด้วยภาษา Java

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Javac 5	4540	79	98.3
JastAddJ5	4569	50	98.9
JGC	4569	50	989

4.2 การทดสอบการทำงานของส่วนขยายของตัวแปลภาษาส่วนหน้าด้วยภาษา Groovy

ในการทดสอบการทำงานของส่วนขยายของตัวแปลภาษานี้ใช้เงื่อนไขและสภาพแวดล้อมเดียวกัน ตัวแปลภาษา JGroovy ที่เรียกว่า JGC นั้นจะต้องถูกประเมินด้วยกรณีทดสอบภาษา Groovy พร้อมกันกับตัวแปลภาษา Groovy ด้วยกรณีทดสอบ 515 กรณีโดยกรณีทดสอบนี้ได้ออกแบบมาจากมาตรฐานภาษา Groovy โดยกรณีทดสอบทั้งหมดไม่รวมกรณีทดสอบที่เป็นภาษา Java

4.2.1 การทดสอบช่องและประโยค (An experiment of Block and statement)

กระบวนการทดสอบนี้ประกอบไปด้วย 8 ส่วนย่อยได้แก่ Groovy-method-declaration, Groovy-method-invocation, Groovy-break-statement, Groovy-continue-statement, Groovy-method-overloading, Groovy-method-parameter, Groovy-return-statement และGroovy-switch-statement โดยมีกรณีทดสอบ 106 กรณี โดยตัวอย่างของกรณีทดสอบสามารถแปลงออกมาเป็นภาษา Groovy ได้ดังนี้

```
class T001G1 {

       public static void main(args){

            def MAX = 10;
            def sum = 0;
            for(java.lang.Object k in 1..MAX) {
                 continue ;
                 sum += value;
            }
            println("sum: ${sum}");
       }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนของ block-and-statement แสดงอยู่ในตารางที่ 4.60

ตารางที่ 4.60 แสดงผลการทดสอบของ Block and statement

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	106	0	100
JGC	106	0	100

4.2.2 การทดสอบโครงสร้างคำต่างๆ (An experiment of Lexical structure)

ในส่วนของการตรวจสอบการแปลงคำนี้มีกรณีทดสอบทั้งหมด 80 กรณีซึ่งครอบคลุมการทดสอบย่อย 3 ส่วนได้แก่ create-class-template, groovy-array และ node-builder โดยสามารถแปลงออกมาเป็นภาษา Groovy ได้ดังต่อไปนี้

```
class T002G1 {

       public static void main(args){

          def file = new File('library.template')

          def books = ['1234567890' : ['Ken Barclay', 'Groovy',
  'Elsevier'],'0750660989' : ['John Savage', 'OOD with UML and
  JAVA', 'Elsevier'],'0130373265': ['Ken Barclay', 'C
  Programming', 'Prentice Hall'] ]

          def writable = new
                SimpleTemplateEngine().createTemplate(file)
          }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนนี้แสดงอยู่ในตารางที่ 4.61

ตารางที่ 4.61 แสดงผลการทดสอบของ Lexical structure

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	80	0	100
JGC	80	0	100

4.2.3 การทดสอบนิพจน์ (An experiment of expression)

เนื่องจากโครงสร้างภาษา Groovy ค่อนข้างคล้ายคลึงกับภาษา Java ทำให้อาจจะใช้กรณีทดสอบร่วมกันได้ แต่เครื่องหมายดำเนินการที่เพิ่มขึ้นมาทำให้ต้องเพิ่มกรณีทดสอบใหม่ อย่างเช่นชุดทดสอบของ string compare expression, regular expression และ pre-and-post-increment-decrement โดยเพิ่มกรณีทดสอบเฉพาะเครื่องหมายดำเนินการที่เพิ่มขึ้น จำนวน 70 กรณีโดยสามารถแปลงออกมาเป็นภาษา Groovy ได้ดังต่อไปนี้

```
class T003G1 {
        public static void main(args){

            def regex = "test"

            def c = 'csfsafheeseffsfdcasdfke' =~ 'cheese'
        }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนของนิพจน์นี้แสดงอยู่ใน ตารางที่ 4.62

ตารางที่ 4.62 แสดงผลการทดสอบของ Expression

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	70	0	100
JGC	70	0	100

4.2.4 การทดสอบรูปแบบไวยากรณ์ธรรมชาติของ List และ Maps (An experiment of Native syntax for List and Maps)

รูปแบบธรรมชาติของ List และ Maps ทำให้สามารถกำหนดและจัดการข้อมูลประเภทนี้ได้อย่างรวดเร็ว การทดสอบความถูกต้องนี้ได้ใช้กรณีทดสอบจำนวน 45 กรณีโดยสามารถแปลงออกมาเป็นภาษา Groovy ได้ดังต่อไปนี้

```
class T004G1 {

        public static void main(args){

            def library = ['Groovy' : ['Ken', 'John'],'OOD' :
                           ['Ken'], 'Java' : ['John', 'Sally'],
                           'UML' : ['Sally'], 'Basic' : ]

            def menu = ['1' : doAddBook,
                        '2' : doRemoveBook,
                        '3' : doLendBook,
              '4' : doReturnBook,
              '5' : doDisplayLoanStock,
              '6' : doDisplayNumberBooksOnLoanToBorrower,
              '7' : doDisplayNumberBorrowersOfBook
             ]
        }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนรูปแบบธรรมชาติของ List และ Maps นี้แสดงอยู่ในตารางที่ 4.63

ตารางที่ 4.63 แสดงผลการทดสอบของ Native syntax for List and Maps

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	45	0	100
JGC	45	0	100

4.2.5 การทดสอบการประกาศตัวแปรและตัวอักษร (An experiment of Variable declaration and Literal)

กรณีทดสอบ 170 กรณีถูกนำมาทดสอบกับรูปแบบการประกาศตัวแปรและตัวอักษรที่เพิ่มขึ้นมาใหม่โดยสามารถแปลงออกมาเป็นภาษา Groovy ได้ดังต่อไปนี้

```
class T005G1 {

        public static void main(args){

          def rhyme = 'Humpty Dumpty sat on a wall';

          println(!(rhyme = ~'[hd]umpty'));
        }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนประกาศตัวแปรและตัวอักษรนี้แสดงอยู่ในตารางที่ 4.64

ตารางที่ 4.64 แสดงผลการทดสอบของ Variable declaration and Literal

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	170	0	100
JGC	170	0	100

4.2.6 การทดสอบรูปแบบปิด (An experiment of Closure)

รูปแบบปิดเป็นวิธีการที่เพิ่มเข้ามาเพื่อจัดการปัญหาเฉพาะทางในรูปแบบธรรมชาติของภาษาเฉพาะทางนั้นๆ โดยมีกรณีทดสอบ 44 กรณีเพื่อตรวจสอบตัวแปลภาษาต่อการทำงานในลักษณะนี้โดยสามารถแปลงออกมาเป็นภาษา Groovy ได้ดังต่อไปนี้

```
class T006G1
  public static void main(args){

        def anyElement = [11, 12, 13, 14].any {element ->
                          element > 12}
          println "anyElement: ${println letter}"
  }
}
```

โดยผลจากการทดสอบการทำงานของตัวแปลภาษาในส่วนรูปแบบปิดนี้แสดงอยู่ในตารางที่ 4.65

ตารางที่ 4.65 แสดงผลการทดสอบของ Closure

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	44	0	100
JGC	44	0	100

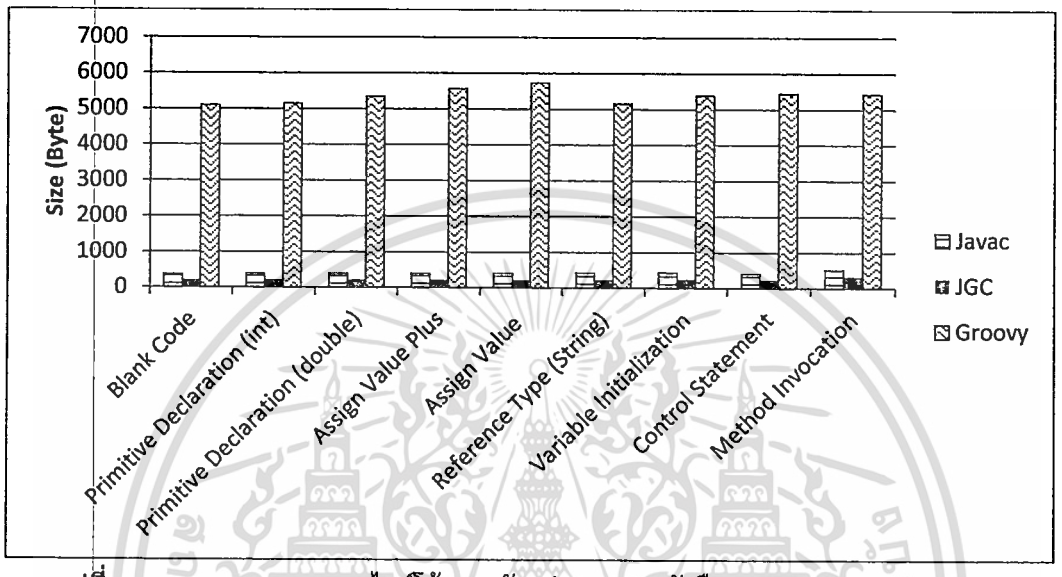
ในการทดสอบนี้เป็นการประเมินการทำงานของตัวแปลภาษาส่วนหน้าของภาษา Groovy ด้วยกรณีทดสอบทั้งสิ้น 515 กรณีโดยผลการทดสอบทั้งหมดแสดงอยู่ในตารางที่ 4.66

ตารางที่ 4.66 แสดงผลการทดสอบของตัวแปลภาษาส่วนหน้าด้วยภาษา Groovy

ตัวแปลภาษา	ทดสอบผ่าน (กรณี)	ทดสอบไม่ผ่าน (กรณี)	คิดเป็นร้อยละ
Groovy	515	0	100
JGC	515	0	100

4.3 การทดสอบการทำงานของตัวแปลภาษาส่วนหลัง

จุดประสงค์ของการขยายความสามารถของตัวแปลภาษาคือตัวแปลภาษานั้นจะต้องสามารถทำงานได้อย่างถูกต้องและสนับสนุนรูปแบบการทำงานต่างๆ รวมถึงสามารถทำงานได้จริงซึ่งในส่วนนี้ได้แบ่งการทดสอบการแปลงรหัสต้นฉบับให้ออกมาเป็นไบต์โค้ดเป็น 2 ส่วนคือ เปรียบเทียบขนาดของไบต์โค้ดที่ได้จากตัวแปลภาษาที่ต่างชนิดกันของภาษา Java และ เปรียบเทียบขนาดของไบต์โค้ดของภาษา Groovy ที่ได้จากตัวแปลภาษา Groovy และ JGC

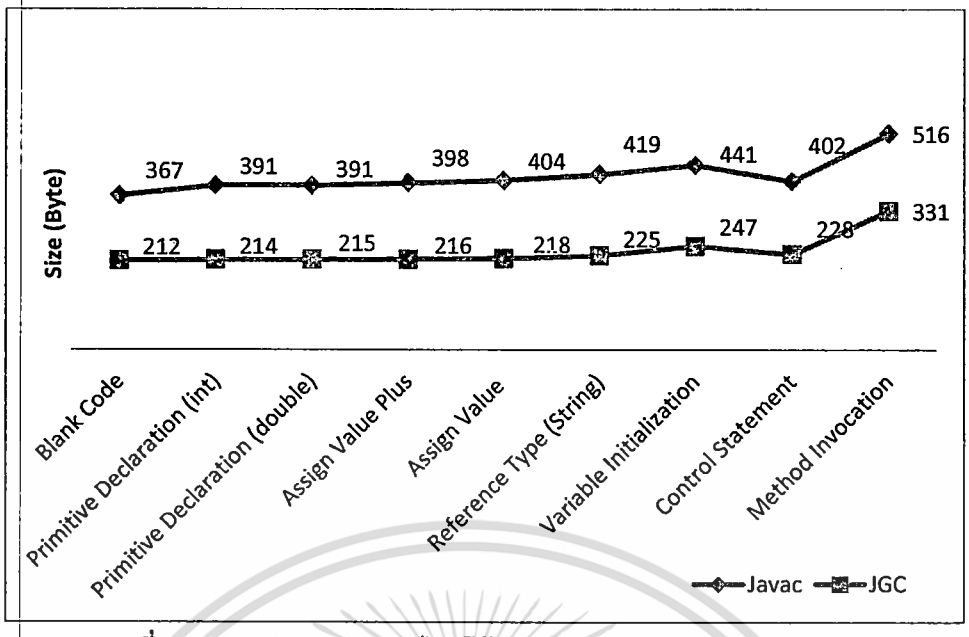


รูปที่ 4.1 แผนภาพแสดงขนาดไบต์โค้ดของตัวแปลภาษา 3 ตัวคือ Javac JGC และGroovy

4.3.1 การวัดขนาดไบต์โค้ดของภาษา Java (Java byte code evaluation)

การทดสอบในส่วนนี้เป็นการวัดขนาดไบต์โค้ดที่ได้จากตัวแปลภาษา Java ทั้ง 3 ตัว ได้แก่ Javac JGC และ Groovy ในกรณีการทำงานต่างๆ ซึ่งมีกราฟแสดงผล 2 ชนิดจากการทดสอบในส่วนนี้โดยกราฟแรก (รูปที่ 4.1) ได้ทำการเปรียบเทียบไบต์โค้ดที่ถูกแปลโดย Javac JGC และ Groovy

เพื่อเปรียบเทียบขนาดของไบต์โค้ดที่ได้จาก Javac และ JGC ให้ชัดเจนรูปที่ 4.2 จึงแสดงขนาดของไบต์โค้ดในรูปแบบตัวเลข

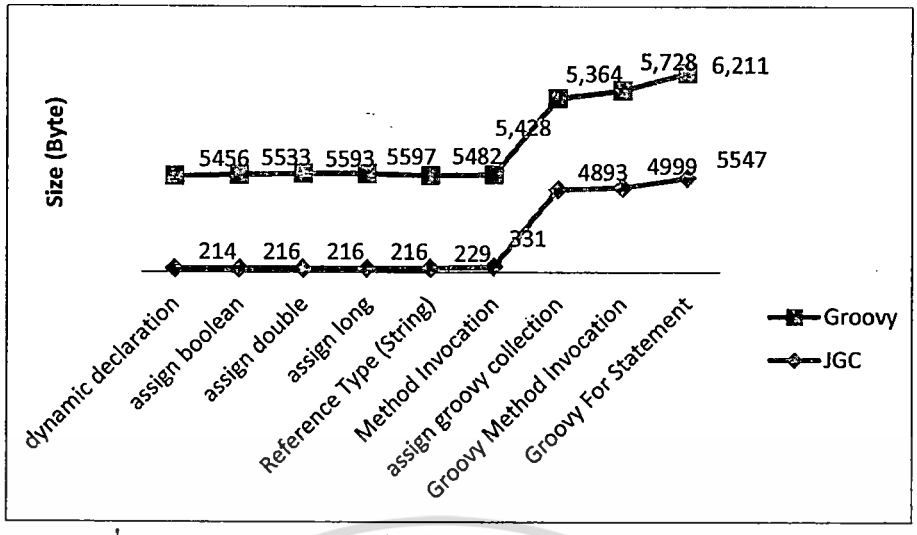


รูปที่ 4.2 แผนภาพแสดงขนาดไบต์โค้ดของตัวแปลภาษา Javac และ JGC

4.3.2 การวัดขนาดไบต์โค้ดของภาษา Groovy (Groovy byte code evaluation)

การทดสอบในส่วนนี้ได้แสดงให้เห็นถึงขนาดของไบต์โค้ดจากตัวแปลภาษา JGC และ Groovy โดยในรูปที่ 4.3 ที่เกิดขึ้นในกรณีการทำงานต่างๆ

จากกราฟสุดท้ายแสดงให้เห็นว่าตัวแปลภาษา Groovy ได้ผลิตไบต์โค้ดที่มีขนาดใหญ่เมื่อทำงานในการทำงานปกติของภาษา Java [68]เนื่องจากว่าภาษา Groovy ได้ทำการสร้าง meta-class เมธอดขึ้นภายในคลาส โดยปกติแล้ว meta-class จะทำงานในช่วง runtime แต่สำหรับงานปกติทั่วไปแล้วการทำงานในส่วนนี้จะไม่ถูกเรียกใช้งานดังนั้นโค้ดดังกล่าวจะไม่ทำประโยชน์หากไม่มีการใช้งาน API ของภาษา Groovy และทำให้การทำงานช้าลงเนื่องจากมีต้นทุนจากโค้ดที่สูงขึ้น [69, 70] โดยจุดภายในกราฟจุดแรกถึงจุดที่หกแสดงให้เห็นถึงการทำงานปกติของภาษา Java ซึ่งเห็นได้ว่าตัวแปลภาษา Groovy สร้างโค้ดที่ไม่จำเป็นออกมา และหลังจากนั้นเมื่อมีการใช้งาน API ของภาษา Groovy ตัวแปลภาษา JGC ได้สร้างไบต์โค้ดส่วนที่จำเป็นออกมาเพื่อใช้งาน API ดังกล่าวทำให้ขนาดเพิ่มขึ้นอย่างมีนัยยะซึ่งทั้ง 2 ใกล้เคียงกัน [71] ขนาดของไบต์โค้ดที่เล็กลงนั้นสำคัญต่อการทำงานในเครื่องมือที่มีทรัพยากรที่จำกัด [72] ดังนั้นไบต์โค้ดที่เล็กลงนั้นจึงมีความสำคัญต่อประสิทธิภาพการทำงาน [73]



รูปที่ 4.3 แผนภาพแสดงขนาดไบต์โค้ดของตัวแปลภาษา Groovy และ JGC

4.4 การทดสอบประสิทธิภาพการทำงานของไบต์โค้ด

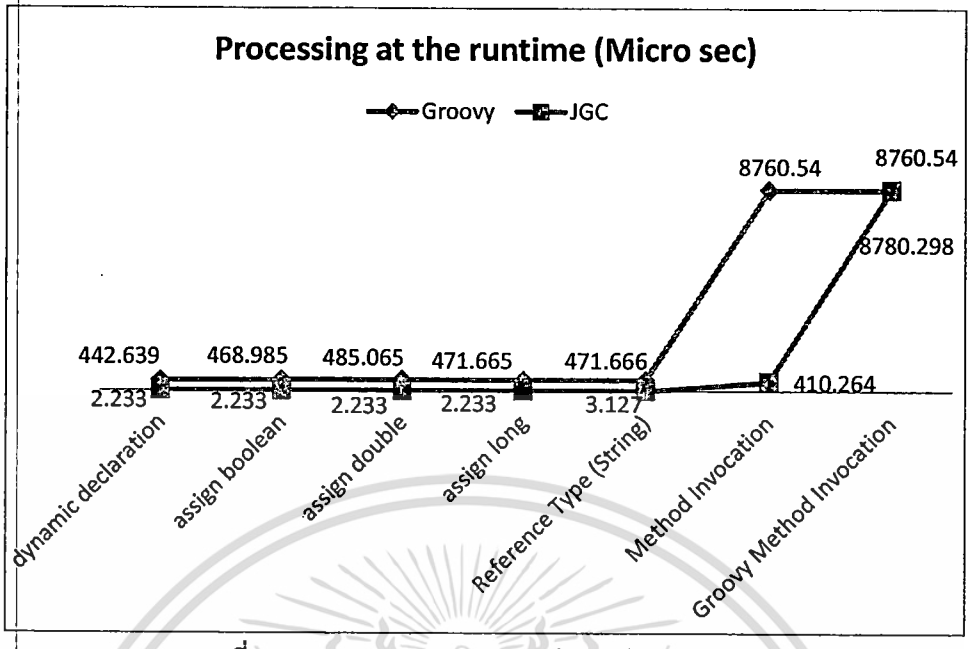
ขนาดของไบต์โค้ดไม่ใช่หน่วยวัดคุณภาพเพียงอย่างเดียวเท่านั้น [74] คุณภาพของไบต์โค้ดเป็นหนึ่งในเรื่องสำคัญที่ต้องได้รับการประเมินและวัดด้วยเช่นกัน นอกจากนี้ปริมาณการใช้หน่วยความจำ [74, 75] ปริมาณการใช้ CPU เป็นตัวชี้วัดที่สำคัญของประสิทธิภาพการทำงานของไบต์โค้ดนี้ด้วย [76] โดยการทดสอบนี้ได้ใช้กรณีการทำงานต่างๆ เพื่อทดสอบการทำงานและเปรียบเทียบไบต์โค้ดที่ได้จากตัวแปลภาษาทั้ง 2 ตัวคือ JGC และ Groovy

4.4.1 การวัดการทำงานช่วง Runtime (Processing at the runtime)

ในการวัดการทำงานนี้ได้ใช้ตัววัดการทำงาน Grande Benchmark เพื่อประเมินเวลาที่ถูกใช้ในการดำเนินการช่วง Runtime [65, 66] โดยใช้วัดกรณีในช่วงการทำงานที่ต่างกันดังนี้

- การประกาศตัวแปรแบบพลวัตร (Dynamic declaration)
- การกำหนดค่าความจริง (Assign a boolean value)
- การกำหนดค่าความ double
- การกำหนดค่าความ long
- การกำหนดค่าอ้างอิง
- การเรียกใช้เมธอดแบบปกติ (Method invocation)
- การเรียกใช้เมธอดแบบภาษา Groovy (Groovy Method invocation)

โดยกราฟที่แสดงในรูปที่ 4.4 ชี้ให้เห็นผลจากการทดสอบการทำงานในงานที่แตกต่างกัน เพื่อให้เห็นคุณภาพของไบต์โค้ดที่ได้จากตัวแปลภาษาทั้ง 2 ชนิด



รูปที่ 4.4 แผนภาพแสดงเวลาการทำงานช่วง Runtime

4.4.2 การวัดการการใช้ CPU และหน่วยความจำ (CPU and Memory usages)

ในส่วนนี้ใช้ในการวัดการทำงานของ CPU และหน่วยความจำที่ถูกใช้ในช่วงเวลาทำงานจริงโดยกรณีทดสอบจะเป็นกรณีเดียวกับการทดสอบก่อนหน้า โดยกราฟที่แสดงในรูปที่ 4.5 ชี้ให้เห็นผลจากการทดสอบการทำงานในงานที่แตกต่างกันเพื่อให้เห็นการใช้ทรัพยากรเครื่องที่ได้จากตัวแปลภาษาทั้ง 2 ชนิด

บทที่ 5
สรุป

5.1 สรุป

ปริญญานิพนธ์นี้ได้นำเสนอแนวทางในการขยายภาษาและตัวแปลภาษา Java ร่วมกับภาษาที่คล้ายคลึงกันเพื่อขยายความสามารถในการแก้ไขปัญหาเฉพาะทาง และรวมไปถึงการขยายความสามารถของภาษาในด้านอื่นๆ เพื่อเพิ่มลักษณะและวิธีการต่างๆ สำหรับการเขียนโปรแกรมสมัยใหม่ โดยได้นำเสนอแนวทางการขยายภาษาผ่านทางภาษาและตัวแปลภาษา JGroovy

ภาษา JGroovy ได้นำภาษา Java5 ตามมาตรฐานของ JLS [19] มาเป็นภาษาหลักและขยายความสามารถของภาษา Groovy เพื่อจัดการด้านปัญหาเฉพาะทางผ่านทางลักษณะการทำงานที่ภาษา Groovy ได้เตรียมลักษณะปิด (closure) ซึ่งสามารถรองรับรูปแบบของภาษาเฉพาะทางใดๆ โดยการออกแบบได้ใช้ตัวคัดแยกคำสถานะจำกัดแบบลำดับชั้น (Stack Machine Analyzer) เพื่อรองรับรูปแบบของภาษาที่คล้ายคลึงกัน และรวมไปถึงวิธีการฉีดรหัสจำลองเข้าสู่รหัสหลัก (Dummy code injection) เพื่อรองรับการสร้างคลาสไฟล์สำหรับโค้ดของภาษาที่ถูกขยายลงไป ลักษณะการขยายดังกล่าวทำให้ไบต์โค้ดที่ได้จากตัวแปลภาษา JGC มีลักษณะที่ได้เปรียบดังนี้

- ไบต์โค้ดมีขนาดเล็กกว่าเมื่อเปรียบเทียบไบต์โค้ดที่ได้จากตัวแปลภาษา Java อย่าง Javac โดยมีขนาดไบต์โค้ดเล็กกว่า 43.74 % ± 3.25 % โดยเฉลี่ย
- ไบต์โค้ดที่ได้มีขนาดเล็กที่สุดเมื่อเปรียบเทียบไบต์โค้ดที่ได้จากตัวแปลภาษาทั้ง 3 ตัว
- ขนาดไบต์โค้ดที่ได้มีขนาดที่เล็กลงเนื่องจากตัดทอนการสร้างไบต์โค้ดที่ไม่จำเป็น โดยสามารถลดขนาดไบต์โค้ดลงได้ 8 - 12 % เมื่อมีการเรียกใช้ API ของภาษา Groovy

วิธีการขยายภาษาสามารถใช้ประโยชน์จากการทำงานของภาษาหลักที่เป็นคุณลักษณะและโครงสร้างพื้นฐานของภาษา ซึ่งวิธีการนี้นอกจากจะขยายความสามารถของภาษาใหม่ลงไปในภาษาหลักแล้วยังช่วยลดข้อด้อยจากภาษาทั้ง 2 ชนิดด้วยการทำงานร่วมกัน โดยการวัดประสิทธิภาพการทำงานของไบต์โค้ดที่ได้จากตัวแปลภาษาทั้ง 3 ตัวด้วยตัววัดวัดประสิทธิภาพอย่าง Grande Benchmark [65, 66] และ VisualVM [67] สำหรับประเมินการเวลาการทำงานของไบต์โค้ดช่วง Runtime ปริมาณการใช้หน่วยความจำและ CPU พบว่า JGC มีลักษณะที่ได้เปรียบดังนี้

- การทำงานของไบต์โค้ดช่วง Runtime สามารถทำงานดีกว่าไบต์โค้ดของตัวแปลภาษา Groovy โดยเฉลี่ย 53.67 %
- ลดปริมาณการใช้ทรัพยากรเครื่องอย่าง CPU ลดลงเมื่อเปรียบเทียบไบต์โค้ดที่ได้จากตัวแปลภาษา Groovy โดยเฉลี่ย 6.31 %
- ลดปริมาณการใช้ทรัพยากรเครื่องอย่างหน่วยความจำลดลงเมื่อเปรียบเทียบไบต์โค้ดที่ได้จากตัวแปลภาษา Groovy โดยเฉลี่ย 34.92 % เมื่อตัดทอนไบต์โค้ดที่ไม่จำเป็นออก และสามารถทำงานได้เท่ากันเมื่อมีการเรียกใช้ API ของภาษา Groovy

ตัวแปลภาษา JGC เป็นตัวแปลภาษาที่ใช้วิธีการขยายความสามารถของภาษา JGroovy ซึ่งสามารถสร้างไบต์โค้ดที่ขนาดที่เล็กลงได้จากการทำงานร่วมกันของคุณลักษณะและโครงสร้างพื้นฐานของภาษาหลักและภาษาที่ขยายความสามารถลงไป วิธีการขยายความสามารถเป็นวิธีการที่เรียบเรียง

และสร้างภาษาและตัวแปลภาษาขึ้นมาใหม่โดยจะใช้ภาษาที่ทำงานได้อย่างมีประสิทธิภาพอย่างภาษา Java โดยตัวแปลภาษาแบบขยายความสามารถนี้จะสนับสนุนโครงสร้างของรหัสต้นฉบับได้เป็นอย่างดี และเมื่อมีการสร้างไบต์โค้ดจะสามารถเลือกวิธีการที่เหมาะสมเพื่อสร้างไบต์โค้ดเพื่อให้ได้ไบต์โค้ดที่มีประสิทธิภาพ กะทัดรัด และเหมาะสมที่จะทำงานบนแพลตฟอร์มของ JVM

5.2 การทำงานในอนาคต

วิศวกรรมตามรอย (Forward Engineering) เป็นวิศวกรรมที่เป็นทฤษฎีพื้นฐานที่สำคัญสำหรับการทำงานของตัวแปลภาษาซึ่งไบต์โค้ดเป็นผลผลิตที่สำคัญจากวิศวกรรมตามรอยของตัวแปลภาษา [77] นอกจากนี้การเพิ่มประสิทธิภาพของไบต์โค้ดยังเป็นหนึ่งในเรื่องสำคัญที่ต้องติดตาม โครงร่างการทำงานของ Soot ได้เตรียมวิธีการเพื่อปรับแต่งการทำงานของไบต์โค้ดและยังเอื้อให้ทำวิศวกรรมแบบย้อนกลับ [78, 79] (Reverse Engineering) ซึ่งลักษณะวิธีการดังกล่าวเหมาะสมกับการทำงานเพื่อเพิ่มศักยภาพของตัวแปลภาษาในอนาคต และหัวข้อสำคัญอีกข้อหนึ่งคือการนำวิธีการขยายความสามารถของภาษาและตัวแปลภาษาในลักษณะนี้ไปใช้ในการขยายความสามารถของภาษาหลักอื่นๆ ต่อไป [80]

และในความเป็นไปได้นอกจากนี้คือการสร้างตัวแปลภาษาแบบพลวัตร การทำงานของการขยายตัวแปลภาษาในลักษณะนี้ใช้วิธีการเขียนโปรแกรมเชิงลักษณะแบบคงที่ (Static Aspect-oriented paradigm) เป็นโครงสร้างหลักในการสร้างตัวแปลภาษา แต่วิธีการเขียนโปรแกรมเชิงลักษณะแบบพลวัตร [81] (Dynamic Aspect-oriented paradigm) นั้นได้ถูกพัฒนาอยู่แล้วดังนั้นการใช้การจัดการแบบพลวัตรต่อการสร้างตัวแปลภาษาจะทำให้สามารถจัดการเปลี่ยนแปลงตัวแปลภาษาได้ในระดับ Runtime ดังนั้นการทำงานแบบพลวัตรจึงเป็นหนึ่งในหัวข้อที่น่าสนใจที่จะทำต่อไปในอนาคตด้วยเช่นกัน

เอกสารอ้างอิง

[1] John C. Reynolds, **"Theories of Programming Languages"**, Cambridge University Press, New York, 1998.

[2] M. Fowler, **"Domain-Specific Languages"**, Addison-Wesley Professional, 2011.

[3] M. Ward, Language Oriented Programming, Software-Concepts and Tools, 1994; 15: 147-161.

[4] J. Bloch, **"Effective Java: Programming Language Guide"**, Addison-Wesley, 2001.

[5] Ian C. Pyle, **"ADA Programming Language"**, Prentice Hall PTR Upper Saddle River, NJ, USA, 1981.

[6] http://www.jruby.org, "JRuby official website".

[7] Keen A K, Tingjian G, Justin T M, Olsson R A, **"JR: Flexible Distributed Programming in an Extended Java"**, ACM Transactions on Programming Languages and Systems (TOPLAS), 2004; 26(3).

[8] J. Liu, C. Myers, **"JMatch: Iterable Abstract Pattern Matching for Java"**, Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages (PADL'03), Lecture Notes in Computer Science, Springer Verlag, 2003; 2562: 110-127.

[9] http://www.groovy.codehaus.org, "Groovy official website".

[10] T. Pittman, **"The Art of Compiler Design: Theory and Practice"**, Prentice Hall, 1991.

[11] Laura L. Dos Reis, Anthony J. Dos Reis, **"Compiler Construction Using Java, Javacc, And Yacc"**, Wiley-Blackwell, 2012.

[12] J. Russell, R. Cohn, **"Generic Eclipse Modeling System"**, Book on Demand, 2012.

[13] N. Kalicharan, **"Pascal Programming - A Beginner's Course"**, Noel Kalicharan, 2006.

[14] M. Dadashzadeh, **"PL / 1 Programming Language Essentials"**, Dover Publications, 2012.

[15] http://www.huskell.org, "Huskell official website".

[16] G. Hutton, **"Programming in Huskell"**, Cambridge University Press", 2007.

[17] R. Harrison, L. G. Samaraweera, M. R. Dobie, P. H. Lewis, **"Comparing Programming Paradigms: an Evaluation of Functional and Object-Oriented Programs"**, 1996.

[18] S. Schach, **"Object-Oriented and Classical Software Engineering Seventh Edition"**, McGraw-Hill, 2006.

[19] J. Gosling, B. Joy, G. Steele, G. Bracha, **"The Java Language Specification Third Edition"**. Addison-Wesley, 2005.

[20] S. J. Chapman **"Fortran 95/2003 for Scientists & Engineers"**, McGraw-Hill Science, 2007.

[21] E. Hopcroft, D. Ullman, **"Introduction to automata theory, languages and computations"**, Addison-Wesley, 1979.

[22] M. Sipser, **"Introduction to the Theory of Computation"**, PWS Publishing, 1997.

[23] D. E. Knuth, **"The genesis of attribute grammars"**, Proceedings of the international conference on Attribute grammars and their applications, LNCS, 1990, vol. 461, p. 1-12.

[24] J. Jones, **"Abstract Syntax Tree Implementation Idioms"**, In: Proceedings of the 10th Conference on Pattern Languages of Programs, 2003.

[25] http://www.spoon-library.com, "Spoon Framework official website".

[26] M. Tatsubori, S. Chiba, M. Killijian, K. Itano, **"OpenJava: A Class-Based Macro System for Java"**, Lecture Notes in Computer Science 1826, Reflection and Software Engineering, , Walter Cazzola, Robert J. Stroud, Francesco Tisato (Eds.), Springer-Verlag, pp.117-133, 2000.

[27] L. Tratt, **"Compile-time meta-programming in a dynamically typed OO language"**, Dynamic Languages Symposium, pages 49-64, October 2005.

[28] N Nystrom, M. R. Clarkson, A. C. Myers, **"Polyglot: An Extensible Compiler Framework for Java"**, Proc. 12th International Conference on Compiler Construction, Warsaw, Poland, April 2003. LNCS 2622, pp. 138–152.

[29] E. Torbjorn , H. Gorel , **"The JastAdd Extensible Java Compiler"**, OOPSLA07, 2007.

[30] L. C. L. Kats, M. Bravenboer, E. Visser, **"Mixing Source and Bytecode. A Case for Compilation by Normalization"**, The 23rd ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2008).ACM SIGPLAN Notices 43(10), pages 91—108. Nashville, Tenessee, USA in October 2008.

[31] E. Bruneton, R. Lenglet, T. Coupaye, **"ASM: a code manipulation tool to implement adaptable systems, Adaptable and extensible component systems"**, November 2002, Grenoble, France

[32] http://commons.apache.org/bcel/, "Apache Commons BCEL official website".

[33] E. Kuleshov, **"Using ASM framework to implement common bytecode transformation patterns"**, AOSD.07, March 2007, Vancouver, Canada.

[34] G. Kiczales, J. Lamping , A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, J. Irwin, **"Aspect-Oriented Programming"**, ECOOP'97, Lecture Notes in Computer Science Springer Verlag, 1997; 1241: 220-242.

[35] M. Bravenboer, E. Visser, **"Concrete Syntax for Objects: Domain-Specific Language Embedding and Assimilation Without Restrictions"**, In OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pages 365–383. ACM, 2004.

[36] A. van Deursen, P. Klint, J. Visser, **"Domain-Specific Languages: an Annotated Bibliography"**, SIGPLAN, 35(6):26–36, 2000.

[37] M. Bravenboer, K. T. Kalleberg, R. Vermaas, E. Visser, **"Stratego/XT 0.17, A Language and Toolset for Program Transformation"**, Sci. Comput. Program, 72(1-2):52–70, 2008.

[38] http:// www.jython.org., "Jython - official website".

[39] D. Phillips, **"Python 3 Object Oriented Programming"**, Packt Publishing Ltd, 2010.

[40] G. R. Andrews, R. A. Olsson,**"The SR Programming Language: Concurrency in Practice"**, Benjamin/Cummings, 1993.

[41] J. Engel, **"Programming for the JavaTm Virtual Machine"**, Addison-Wesley Professional, 1999.

[42] R. Grimm, **"Better extensibility through modular syntax"**, PLDI '06, pp. 38–51, June 2006.

[43] P. Gazzillo, R. Grimm, **"SuperC: Parsing all of C by taming the preprocessor"**, LDI '12, pp. 323–334, June 2012.

[44] http://projectfortress.java.net/, "Project Fortress - official website".

[45] M. Hirzel, N. Nystrom, B. Bloom, J. Vitek, **"Matchete: Paths through the pattern matching jungle"**, Practical Aspects of Declarative Languages (PADL08), 2008.

[46] G. Hedin, M. Mernik, **"Reference Attributed Grammars"**, in D.Parigot Second Workshop on Attribute Grammars and their Applications. WAGA'99, 1999; 153-172.

[47] G. Hedin, **"Reference Attribute Grammar"**, Informatica, 2000; 24(3): 301-317.

[48] M. Zenger, M. Odersky, **"Extensible Algebraic Datatypes with Defaults"**, International Conference on Functional Programming, Firenze, September 2001.

[49] http://www.fosd.de/fuji "Fuji - official website".

[50] R. Bellman, **"Dynamic Programming"**, Dover Publications; Reprint edition, 2003.

[51] D. Flanagan, **"The Ruby Programming Language"**, O'Reilly Media; First Edition, 2008.

[52] A. Goldberg, D. Robson, **"SmallTalk 80"**, Addison-Wesley Professional, 1989.

[53] G. Kiczales, J. D. Rivieres, D. G. Bobrow, **"The Art of the Metaobject Protocol"**, MIT Press, 1991.

[54] C. H. Roth, L. L. Kinney Jr, **"Fundamentals of Logic Design"**, Thomson-Engineering, 2009.

[55] S. Sateanpattanakul, A. Walairacht, **"JGroovy: An Extensible Programming Language with Groovy"**, ICACT2010. 2010.

[56] N. Chomsky, **"Three Models for the Description of Language"**, IRE Transactions on Information Theory 2 (2): 113–123, 1956.

[57] T. Ekman, G. Hedin, **"Reusable Language Specification Modules in JastAdd II"**. Workshop on Evolution and Reuse of Language Specifications for DSLs, ERLS, 2004.

[58] K. A. Wright, M. Felleisen, **"A Syntactic Approach to Type Soundness"**, Information and Computation, 1992; 1(115): p. 33-94.

[59] S. Stelting, O. Maassen, **"Applied Java Patterns"**, Prentice Hall, 2001.

[60] R. Vallee-Rai, L. H. Sundaresan, P.V. Lam, E. Gagnon, **"Soot a Java optimization framework"**, In Proceedings of CASCON99. IBM Press, 1999.

[61] R. L. Clausen, **"A Java bytecode optimizer using side-effect analysis"**, Concurrency:Practice & Experience, 1997; 9(11):1031-1045.

[62] T. Lindholm, F. Yellin, **"The Java Virtual Machine Specication"**, Second Edition. Addison-Wesley, 1999.

[63] http://www.oracle.com, "Java official website".

[64] http://www.sources.redhat.com/mauve, "Jack testsuite official website".

[65] L. A. Smith, J. M. Bull, J. Obdrzálek, **"A parallel java grande benchmark suite"**, Proceedings of the 2001 ACM/IEEE conference on Supercomputing, 2001.

[66] http://www.epcc.ed.ac.uk/javagrande, "Java Grande official website".

[67] http://www. visualvm.java.net, "VisualVM - Javaofficial website".

[68] S. Sateanpattanakul, A. Walairacht, **"JGroovy: An experimental of extensible Java compiler"**, CCCA2011, 2011.

[69] T. Suganuma,T. Ogasawara, M. Takeuchi, T. Yasue , M. Kawahito,K. shizaki, H. Komatsu , T. Nakatani, **"Overview of the IBM Java Just-in-Time compiler"**, IBM System Journal, 2000; 39(1): 175-193.

[70] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, T. Nakatani, **"A Dynamic Optimization Framework for a Java Just-ln-Time Compiler"**, In

Proceeding of Object-Oriented Programming, System Languages, and Application (OOPSLA '01), 2001; 180-194.

[71] S. Sateanpattanakul, A. Walairacht, **"JGroovy: An experimental of extensible Java compiler"**, CCCA2011, 2011.

[72] L. R. Clausen, U. P. Schultz, C. Consel, G. Muller, **"Java Bytecode Compression for Low-End Embedded Systems"**, ACM Transactions on Programming Languages and Systems (TOPLAS), 2000; 22: 471-489.

[73] N. R. Brisaboa, A. Farina , G. Navarro, M. F. Esteller, **"An optimized compression code for natural language text databases"**, In M. A. Nascimento. Editor. Proc. Symp. String Processing and Information Retrieval, Springer Verlag, 2003; 2857: 122-136.

[74] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann, **"The DaCapo benchmarks: Java benchmarking development and analysis"**, In OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications, pages 191–208, Portland, OR, USA, October 2006. ACM Press.

[75] H. B. Lee, D. V. Dincklage, A. Diwan, J. E. B. Moss, **"Understanding the behavior of compiler optimizations"**, Software: Practice and Experience, 36(8):835–844, July 2006.

[76] W. Binder, J. Hulaas, **"Using Bytecode Instruction Counting as Portable CPU Consumption Metric"**, Electronic Notes in Theoretical Computer Science - ENTCS, vol. 153, no. 2, pp. 57-77, 2006.

[77] V. Raja, K. J. Fernandes, **"Reverse Engineering: An Industrial Perspective"**, Springer. 2007.

[78] G. Hoglund, G. McGraw, **"Exploiting Software_ How to Break Code"**, Addison-Wesley Professional, 2004

[79] E. Eilam, E. J. Chikofsky, **"Reversing: secrets of reverse engineering"**, John Wiley & Sons, 2007.

[80] E. J. Chikofsky, J. H. Cross, **"Reverse Engineering and Design Recovery: A Taxonomy"**, IEEE Software 7 (1): 13–17, 1990.

[81] J. Bonér, **"AspectWerkz – Dynamic AOP for Java"**, Proceeding of the 3rd International Conference on Aspect-Oriented Software Development, 2004.

งานวิจัยที่ได้รับการเผยแพร่ในระดับนานาชาติ

[1] S. Sateanpattanakul, A. Walairacht, JGroovy: An Extensible Programming Language with Groovy. ICACT2010. 2010.

[2] S. Sateanpattanakul, A. Walairacht, JGroovy: An experimental of extensible Java compiler. CCCA2011, 2011.

[3] S. Sateanpattanakul, H. Kazuhiko, A. Walairacht, JGroovy: An alternative approach to implement extensible Java compiler, IEEJ Transection On Electrical and Electronic Engineering, Vol. 8 / No. 4 (July 2013 Issue).

ภาคผนวก

จากตารางที่ 3.11 คำสำคัญที่เหลือทั้งหมดเป็นดังนี้

“class”, “this”, “super”, “long”, “int”, “short”, “byte”, “char”, “float”, “double”, “boolean”, “abstract”, “break”, “catch”, “for”, “volatile”, “void”, “try”, “transient”, “throws”, “throw”, “synchronized”, “switch”, “strictfp”, “static”, “return”, “public”, “protected”, “private”, “package”, “native”, “interface”, “instanceof”, “import”, “implementes”, “if”, “goto”, “finally”, “final”, “extends”, “else”, “do”, “default”, “continue”, “const”, “while”

จากตารางที่ 3.13, 3.14, 3.15 ,3.26 และ 3.31 เครื่องหมายที่เหลือทั้งหมดเป็นดังนี้

“=”, “>”, “<”, “!”, “~”, “?”, “:”, “==”, “<=”, “>=”, “!=”, “&&”, “||”, “++”, “--”, “+”, “-”, “*”, “/”, “&”, “|”, “^”, “%”, “<<”, “>>”, “>>>”, “+=”, “-=”, “*=”, “/=”, “&=”, “|=”, “^=”, “%=”, “<<=”, “>>=”, “>>>=”

จากตารางที่ 3.20 คำสำคัญที่เหลือทั้งหมดเป็นดังนี้

“instanceof”, “in”, “int”, “long”, “short”, “byte”, “char”, “float”, “double”, “boolean”, “new”, “;”, “[”, “]”, “(”, “)”, “null”, “true”, “false”, “class”, “this”, “super”, “final”

จากตารางที่ 3.23 และ 3.26 ค่าคงที่ต่างๆ ที่เหลือทั้งหมดเป็นดังนี้

DecimalNumeral, HexNumeral, OctalNumeral, FloatingPointLiteral, GroovyFloatingPointLiteral, ExponentPart, HexadecimalFloatingPointLiteral, “\b”, “\t”, “\n”,, “\f”, “\r”

จากตารางที่ 3.24 คำสำคัญที่เหลือทั้งหมดเป็นดังนี้

“long”, “int”, “short”, “byte”, “char”, “float”, “double”, “abstract”, “break”, “catch”, “for”, “volatile”, “void”, “try”, “transient”, “throws”, “throw”, “synchronized”, “switch”, “strictfp”, “static”, “return”, “public”, “protected”, “private”, “package”, “native”, “interface”, “instanceof”, “import”, “implementes”, “if”, “goto”, “finally”, “final”, “extends”, “else”, “do”, “default”, “continue”, “const”, “while”, “true”, “false”, “null”,

จากตารางที่ 3.25 คำสำคัญที่เหลือทั้งหมดเป็นดังนี้

“def”, “final”, “long”, “int”, “short”, “byte”, “char”, “float”, “double”, “boolean”, “null”, “true”, “false”, “class”, “this”, “super”

ประวัติผู้เขียน

ชื่อ-นามสกุล นายศิวดล เสถียรพัฒนากูล
วัน เดือน ปีเกิด 8 มีนาคม 2524 ที่อำเภอ บ้านหมี จังหวัด ลพบุรี
ที่อยู่ ก.15/1 ถ.สวรรค์วิถี ตำบล ปากน้ำโพ อำเภอ เมือง
จังหวัด นครสวรรค์ 60000 โทร.056-214247
ประวัติการศึกษา 2542 วิศวกรรมศาสตรบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยเทคโนโลยีสุรนารี
2549 วิศวกรรมศาสตรมหาบัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยเทคโนโลยีสุรนารี
ความชำนาญเฉพาะด้าน 1.) วิศวกรรมซอฟต์แวร์
2.) การออกแบบภาษาและตัวแปลภาษาคอมพิวเตอร์

ประสบการณ์การทำงานและผลงานวิจัย
ปัจจุบัน อาจารย์คณะวิศวกรรมศาสตร์ สาขาวิชาวิศวกรรมคอมพิวเตอร์
มหาวิทยาลัยเกษตรศาสตร์ วิทยาเขตกำแพงแสน