บทที่ 9

วากยสัมพันธ์

9.1 การนิยามภาษาโปรแกรม

9.2 วากยสัมพันธ์

9.3 รูปแบบของการเขียนไวยากรณ์

9.4 เอกสารอ้างอิงและเว็บไซต์ที่ควรรู้

วัตถุประสงค์

- แนะนำการนิยามภาษาโปรแกรม
- วากยสัมพันธ์และความหมาย
- การเขียนไวยากรณ์ในรูปแบบต่างๆ

หลักการในการเรียนรู้ภาษาธรรมชาติและการเรียนรู้ภาษาโปรแกรมก็มีลักษณะคล้ายคลึงกัน นั่นคือ ผู้เรียนจะต้องรู้โครงสร้างของภาษา รูปแบบของไวยากรณ์ คำศัพท์และความหมายต่างๆ ในภาษานั้นๆ จึงจะสามารถใช้ภาษาได้อย่างถูกต้อง

9.1 การนิยามภาษาโปรแกรม (Language definitions)

สำหรับภาษาโปรแกรมก็ต้องมีการนิยามภาษาโดยระบุในรูปของวากยสัมพันธ์และความหมายเช่นกัน ผู้ที่จะนำนิยามของภาษามาใช้ก็คือ ผู้ออกแบบภาษา ผู้พัฒนาภาษา และโปรแกรมเมอร์

เมื่อเรานึกถึงโปรแกรม เราก็มักจะนึกถึงในแง่รูปแบบ (เขียนอย่างไร) และความหมาย (จะเกิดอะไรขึ้นเมื่อรันโปรแกรม) ของโปรแกรม เราเรียกรูปแบบของโปรแกรมนี้ว่า Syntax และความหมายของโปรแกรมว่า Semantics คอมไพเลอร์จะทำหน้าที่ตรวจสอบข้อผิดพลาดทาง syntax และตรวจสอบข้อผิดพลาดเกี่ยวกับชนิดและการประกาศของโปรแกรม ในขณะที่ข้อผิดพลาดที่เกิดขึ้นขณะรันโปรแกรม เช่น การหารด้วย 0 ไม่ได้จะเป็นเรื่องของ semantic

Syntax หรือวากยสัมพันธ์ จะอธิบายถึงโครงสร้าง และรูปแบบของภาษา ว่าจะสร้างโปรแกรมขึ้นได้อย่างไร ส่วนความหมายของโปรแกรมก็จะอธิบายโดยใช้ Semantics ตัวอย่างเช่น

Syntax ของการเขียนวันที่ สามารถเขียนให้อยู่ในรูปแบบดังนี้

DD/DD/DDDD

โดยใช้ตัวอักษร D แทนตัวเลข ร่วมกับสัญลักษณ์ / ที่ใช้คั่นระหว่างตัวเลข

Semantic ของการเขียนวันที่รูปแบบนี้ จะบอกว่าตำแหน่งของตัวเลขหมายถึงอะไร เช่น ตำแหน่งตัวเลขสองตัวแรกหมายถึงวันที่ สองตัวถัดมาหลังเครื่องหมาย / หมายถึงเดือน และสี่ตัวหลังหมายถึงปี

ตัวอย่างการใช้งาน โดยระบุวันที่ตามรูปแบบที่กำหนด เช่น

25/02/2552

เป็นการเขียนได้ถูกต้องตามหลักไวยากรณ์ และมีความหมายคือ วันที่ 25 เดือนกุมภาพันธ์ ปี พ.ศ. 2552 เป็นต้น

การอธิบายภาษาใดๆ เพื่อให้ผู้ใช้เรียนรู้และนำภาษาไปใช้ได้ สามารถทำได้หลายรูปแบบ ดังนี้

- แบบเรียน (Tutorials) โดยมากจะทำอธิบายโครงสร้างของภาษา และความหมายเมื่อนำไปใช้ใช้งาน โดยยกตัวอย่างประกอบ เพื่อให้เข้าใจได้ง่าย ค่อยๆ สอดแทรก syntax และ semantics ไปด้วยทีละน้อย

- คู่มืออ้างอิง (Reference manuals) เป็นการอธิบาย syntax และ semantics โดยอธิบายจะเน้นที่การอธิบาย syntax อย่างเป็นทางการ ในส่วนของ semantic จะใช้การอธิบายแบบไม่เป็นทางการ โดยใช้คำอธิบายร่วมกับตัวอย่างประกอบเพื่อให้เข้าใจกฎเกณฑ์ของภาษา
- การนิยามแบบเป็นทางการ (Formal definition) เป็นการอธิบาย syntax และ semantics อย่างเป็นทางการในรูปแบบสัญลักษณ์ที่จะไม่ทำให้เกิดการตีความที่กำกวมหรือสับสนอันเนื่องมา จากการอธิบายโดยใช้ภาษาเขียนทั่วไป จุดประสงค์เพื่อให้ผู้เชี่ยวชาญโดยเฉพาะนำไปใช้งาน

9.2 วากยสัมพันธ์ (Syntax)

หากจะให้นิยามของวากยสัมพันธ์อาจกล่าวได้ว่า วากยสัมพันธ์ของภาษาโปรแกรม คือ การอธิบายวิธีการเขียนโปรแกรมที่ถูกต้องตามไวยากรณ์ไว้อย่างชัดเจน เราอาจมองว่า syntax ก็เซตของกฎ ซึ่งคล้ายกับกฎในภาษาธรรมชาติ เช่น กฎหรือ syntax ในการเขียนภาษาอังกฤษที่ระบุว่า เมื่อจบประโยคจะต้องมีเครื่องหมายปิดท้ายประโยคเสมอ ซึ่งอาจเป็นเครื่องหมายจุด เครื่องหมายตกใจ หรือเครื่องหมายคำถาม

การอธิบาย syntax ของภาษาให้ชัดเจน ถูกต้อง ไม่คลุมเครือ ถือเป็นสิ่งสำคัญที่ช่วยทำให้ผู้เขียนคอมไพเลอร์และโปรแกรมเมอร์ทำงานได้ง่ายขึ้น

การอธิบาย syntax ของภาษาสามารถทำได้ในรูปของ *ไวยากรณ์ (grammar)* ไวยากรณ์ คือ ภาษาที่ใช้ในการอธิบายภาษา (language-description-language) หรือเรียกว่า *metalanguage* ซึ่งใช้ในการกำหนดสตริงที่สามารถประกอบกันเป็นโปรแกรมที่ถูกต้องได้ ไวยากรณ์เป็น metalanguage ที่พัฒนาจากทฤษฎีของนักภาษาศาสตร์ชื่อ Noam Chomsky ซึ่งกำหนดไวยากรณ์ออกเป็น 4 ระดับ คือ Regular, Context-free, Context-sensitive และ Unrestricted ซึ่งในที่นี้เราจะกล่าวถึงระดับ C*ontext-free grammars (CFG)*

9.3 รูปแบบการเขียนไวยากรณ์

CFG เป็นเซตของ *production P* เซตของ *terminal symbol T* เซตของ *nonterminal symbol N* และเซตของ *start symbol S*

Syntax ของ production จะอยู่ในรูป $A \rightarrow \omega$ โดย A เป็น *nonterminal symbol* และ ω เป็นกลุ่มของ *nonterminal symbol* และ *terminal symbol*

รูปแบบหนึ่งของ CFG ที่นิยมใช้ในการกำหนด syntax ของภาษาโปรแกรม คือ Backus-Naur Form หรือเรียกโดยย่อว่า BNF ในปี 1960 John Backus และ Peter Naur ได้นำเอาทฤษฎีของ Chomsky มาประยุกต์ใช้ในการอธิบาย syntax ของภาษา Algol ในรูปแบบที่เป็นทางการ จึงเป็นที่มาของชื่อ BNF ในตำราหลายๆ เล่มจะพบว่ามีการใช้ BNF แทนคำว่า CFG จึงอาจกล่าวได้ว่าทั้งสองคำนี้มีความหมายเช่นเดียวกัน

Backus-Naur Form (BNF)

BNF เป็น production ของไวยากรณ์ที่อยู่ในรูป $A \rightarrow \omega$ หรือกฎที่ใช้ในการเขียน เพื่อนำมาใช้ในการกำหนด syntax ของภาษาโปรแกรม

กฎจะประกอบด้วยสองส่วนหลักคือ สัญลักษณ์ A หรือด้านซ้ายมือ (left-hand side – LHS) ซึ่งเป็น *nonterminal symbol N* จะหมายถึงกลุ่มของ *Identifier*, *Integer*, *Expression*, *Statement* และ *Program* โดยมี *start symbol S* ที่หมายถึงโครงสร้างหลักของภาษา และใช้ในการกำหนด production แรก (โดยทั่วไปคือ *Program*) และส่วนที่สองคือ ω หรือด้านขวามือ (right-hand side – LHS) ซึ่งประกอบด้วย *terminal symbol T* คือ ตัวอักษรพื้นฐานที่ประกอบกันเป็นโปรแกรม และ *nonterminal symbol N*

เพื่อให้เข้าใจได้ง่ายขึ้น จะยกตัวอย่างการเขียน BNF ในการกำหนด syntax ของ *binaryDigit*

binaryDigit → 0

binaryDigit → 1

จากตัวอย่างเป็นการกำหนดว่า *binaryDigit* มีค่าเป็น 0 หรือ 1 เท่านั้น *nonterminal symbol* คือ สัญลักษณ์ทุกตัวที่ปรากฏทางด้านซ้ายมือของ production ซึ่งในที่นี้มีเพียงตัวเดียวคือ *binaryDigit* ส่วน *nonterminal symbol* คือสัญลักษณ์อื่นๆ ที่เหลือที่ปรากฏใน production ซึ่งจากตัวอย่างนี้คือ 0 และ 1 และจาก production ทั้งสองข้างต้น เราสามารถเขียนให้อยู่ในรูปที่สั้นลงโดยให้เหลือเพียง production เดียวได้ดังนี้

binaryDigit → 0 | 1

โดยใช้เครื่องหมาย | แทนความหมาย “หรือ” เราเรียกสัญลักษณ์ → และ | ว่าเป็น *metasymbols* คือ เป็นสัญลักษณ์ที่เป็นส่วนหนึ่งของ metalanguage แต่ไม่ได้เป็นส่วนหนึ่งของภาษาที่ถูกกำหนด

ด้านขวามือของ BNF production อาจเป็นลำดับใดๆ ของ *nonterminal symbol* และ *terminal symbol* ตัวอย่างเช่น การกำหนดไวยากรณ์ใช้กับ *Integer* โดยกำหนดว่าเป็นลำดับของ *Digit*s ดังนี้

Integer → *Digit* | *Integer Digit*

Digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

จากตัวอย่างจะเห็นว่า production ที่สองเป็นการกำหนดเลขฐาน 10 ว่าสามารถมีค่าได้ตั้งแต่ 0 ถึง 9 ส่วน production แรกเป็นการกำหนด *Integer* ว่าอาจเป็น *Digit* เพียงตัวเดียว หรือเป็น *Integer* ตามด้วย *Digit* ก็ได้ จะเห็นว่าการกำหนดแบบนี้เป็นลักษณะที่วนซ้ำโดยเรียกตัวเอง (recursion) นั่นคือ *Integer* จะประกอบด้วยตัวเลข *Digit* อย่างน้อย 1 ตัวหรือมากกว่าก็ได้

Derivation

ในการพิจารณาว่าสตริงของสัญลักษณ์หนึ่งจัดอยู่ในกลุ่มไวยากรณ์หนึ่งๆ ทำได้โดยการ derivation หรือเขียนให้อยู่ในรูปของ Parse tree ซึ่งเป็นการตรวจสอบว่าสตริงนั้นได้มาจากการใช้กฎหรือ production ของกลุ่มไวยากรณ์นั้นๆ ดังแสดงในตัวอย่างต่อไปนี้

พิจารณาว่า 352 เป็น *Integer* หรือไม่

เราสามารถทำ derivation ได้โดยกฎของไวยากรณ์ดังต่อไปนี้

1. เริ่มต้นด้วย start symbol *Integer*
2. แทนค่า *Integer* ด้วย *Integer Digit* (ใช้กฎที่สองทางขวามือของ production แรก)
3. แทนค่า *Integer* ตัวแรกด้วย *Integer Digit* อีกครั้ง
4. แทนค่า *Integer* ด้วย *Digit* (ใช้กฎที่หนึ่งทางขวามือของ production แรก)
5. แทนค่า *Digit* ตัวแรกด้วยเลข 3 (ใช้กฎของ production ที่สอง)
6. แทนค่า *Digit* ตัวที่สองด้วยเลข 5 (ใช้กฎของ production ที่สอง)
7. แทนค่า *Digit* ตัวสุดท้ายด้วยเลข 2 (ใช้กฎของ production ที่สอง)

ซึ่งสามารถเขียนอยู่ในรูปลำดับการทำ derivation (ใช้สัญลักษณ์ => ในการกระจายแต่ละขั้น) ได้ดังนี้

Integer => *Integer Digit*
=> *Integer Digit Digit*
=> *Digit Digit Digit*
=> 3 *Digit Digit*
=> 3 5 *Digit*
=> 352

ดังนั้น จึงพิสูจน์ได้ว่า 352 เป็น *Integer* การทำ derivation ข้างต้นเป็นการทำแบบจากซ้ายไปขวา หรือ leftmost derivation เราสามารถทำวิธีอื่นได้เช่นกัน ดังตัวอย่างต่อไปนี้เป็นการทำจากขวาไปซ้ายหรือ rightmost derivation

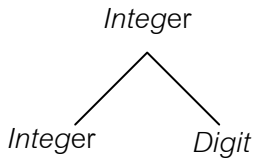
Integer => *Integer Digit*
=> *Integer* 2
=> *Integer Digit* 2
=> *Integer* 5 2
=> *Digit* 5 2
=> 352

Parse tree

นอกจากการทำ derivation แล้วเราอาจใช้วิธีการเขียนให้อยู่ในรูปของ Parse tree เพื่อแสดงว่าสตริงหนึ่งเป็นส่วนหนึ่งของภาษาที่กำหนดไว้ในรูปของ BNF โดยการแสดงเป็นภาพซึ่งมีลักษณะเหมือนต้นไม้ได้ แต่ละขั้นตอนในการทำ derivation ก็คือการสร้างต้นไม้ย่อยนั่นเอง ดังแสดงตัวอย่างขั้นตอน derivation ต่อไปนี้

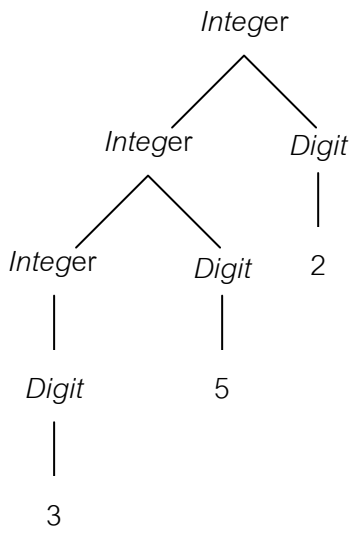
Integer => *Integer Digit*

สามารถเขียนในรูปต้นไม้ย่อยได้ดังรูป



รากของต้นไม้ย้อยใน parse tree คือ โหนดที่เขียนแทน left-hand side หรือ nonterminal ซึ่งสามารถแทนที่ได้ในการทำ derivation ลำดับของลูกจากซ้ายไปขวา คือ right-hand side ของ production ที่ใช้แทนที่

ตัวอย่างการเขียน parse tree ของ 352 ซึ่งเป็น *Integer*



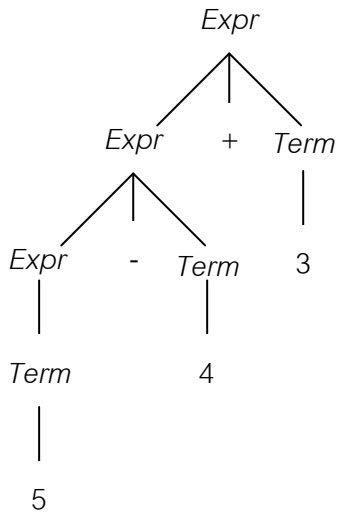
Parse tree มีลักษณะเฉพาะที่เด่นหลายอย่าง ประการแรก โหนดรากของต้นไม้จะเป็น start symbol ของไวยากรณ์เสมอ (จากตัวอย่าง คือ *Integer*) สอง โหนดภายในทุกโหนดจะเป็น nonterminal (จากตัวอย่าง คือ *Integer* และ *Digit*) ซึ่งจำนวนโหนดภายในนี้จะเท่ากับจำนวนขั้น (=>) ของการทำ derivationเสมอ (จากตัวอย่าง คือ 6 โหนด) สาม โหนดภายในแต่ละตัวจะปรากฏตามลำดับจากซ้ายไปขวาเหมือนเช่น right-hand side ของกฎไวยากรณ์เสมอ และสุดท้าย โหนดลูก (leaves) ของต้นไม้จะต้องเป็น terminal symbol เสมอ การอ่านโหนดลูกจะทำจากซ้ายไปขวาก็คือสตริงที่ทำการตรวจสอบนั่นเอง (จากตัวอย่างคือ 352)

กล่าวโดยสรุปได้ว่า การทำ derivation เป็นรูปแบบเชิงเส้นอย่างง่ายของ parse tree ถ้าโครงสร้างของไวยากรณ์ไม่ซับซ้อนก็มักจะจะทำการ derivation แต่ถ้ามีความซับซ้อนมากขึ้นจึงจะนิยมใช้ parse tree

พิจารณาตัวอย่างไวยากรณ์ซึ่งใช้ในการกำหนดภาษาของนิพจน์ทางคณิตศาสตร์ที่มี operator บวก (+) และลบ (–) โดยกระทำกับ operand ซึ่งเป็น Integer ที่เป็นเลขตัวเดียว

Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*

Term → 0 | ….| 9 | (*Expr*)



จากไวยากรณ์นี้ สามารถเขียน parse tree ของสตริง 5 – 4 + 3 ได้ดังรูป จากโครงสร้างนี้ เมื่อทำการตีความจากซ้ายไปขวาของโหนดลูกจะสามารถเขียนได้แบบเดียวคือ (5 – 4) + 3 ซึ่งการตีความนี้ให้ผลที่ต่างจาก 5 - (4 + 3) ถ้าหากไวยากรณ์ใดสามารถตีความหรือสร้าง parse tree ได้มากกว่าหนึ่งแล้วจะถือว่าไวยากรณ์นั้นมีความกำกวม (Ambiguous grammars)

Extendedd BNF (EBNF)

การเขียนไวยากรณ์ในรูปแบบ BNF สามารถเขียนให้อยู่ในรูปที่สั้น กะทัดรัด ถือเป็นรูปแบบโดยย่อของ BNF เรียกว่า Extended BNF หรือ EBNF

พิจารณาตัวอย่างของการเขียน BNF ต่อไปนี้

Expr → *Expr* + *Term* | *Expr* – *Term* | *Term*

Term → 0 | ….| 9 | (*Expr*)

เราสามารถเขียนโดยย่อได้โดยการใช้ metasymbols ในการทำซ้ำ ทางเลือก และตัวเลือกได้ เริ่มจากกฎข้อแรกที่กำหนด Expr ว่าเป็นชุดของ Term ตั้งแต่หนึ่งตัวขึ้นไปที่คั่นด้วยเครื่องหมาย + หรือ - ใน EBNF จะอนุญาตให้กำจัดการเรียกซ้ำตัวเองทางด้านของกฎ จะได้กฎใหม่ดังนี้

Expr → *Term* { (+ | –) *Term* }

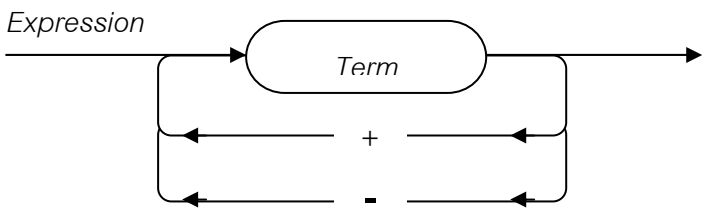
จากกฎใหม่นี้จะเห็นว่ามี metasymbol เพิ่มขึ้นคือ { } ซึ่งใช้บ่งบอกการปรากฏของสัญลักษณ์ที่อยู่ภายในเครื่องหมายว่าอาจจะไม่มีหรือมีมากกว่าหนึ่งก็ได้ และ ( ) ใช้บ่งบอกตัวเลือกที่เป็นไปได้ทั้งหมด ซึ่งจะต้องเลือกว่าเป็นตัวใดตัวหนึ่งที่อยู่ภายในเครื่องหมายเท่านั้น นอกจากนี้ยังมีเครื่องหมาย [ ] ที่ใช้บ่งบอกถึงลำดับของสัญลักษณ์ซึ่งเป็นทางเลือก ซึ่งอาจจะเขียนหรือไม่ก็ได้ ตัวอย่างเช่น ในไวยากรณ์ของ if ดังต่อไปนี้

IfStatement → `if` (*Expr*) *Statement* [`else` *Statement*]

Syntax Diagram

การเขียนไวยากรณ์ในรูปแบบ EBNF สามารถเขียนให้อยู่ในอีกรูปแบบหนึ่งได้ เรียกว่า syntax diagram ซึ่งช่วยในการอธิบาย syntax ของภาษาได้ชัดเจนขึ้นโดยใช้ภาพเป็นสื่อ ทำให้เข้าใจได้ง่ายขึ้น นิยมนำไปใช้ในการสอนรายวิชาการเขียนโปรแกรมเบื้องต้น ที่มักใช้ภาษา Pascal ในการสอน

ตัวอย่าง syntax diagram ของนิพจน์ทางคณิตศาสตร์ดังปรากฏในตัวอย่างของ BNF และ EBNF ก่อนหน้านี้ สามารถแสดงได้ดังรูป



9.4 เอกสารอ้างอิงและเว็บไซต์ที่ควรรู้

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

สุจิตรา อดุลย์เกษม. เอกสารประกอบการสอนองค์ประกอบคอมพิวเตอร์และภาษา.