204433 การแปลภาษาโปรแกรม

การ scan และการ parse expression ทางคณิตศาสตร์

ตัวอย่างของ expression ทางคณิตศาสตร์

(2 * 3) + 4 - 51 % 7

expression นี้มีค่าเท่ากับ 8

ในการคำนวณค่า expression นี้ เราคำนวณสิ่งที่อยู่ในวงเล็บก่อนเพราะถือว่ามี precedence สูงที่สุด จากนั้นก็คำนวณ % operation เราจะได้เห็นการใช้ grammar (pushdown automata) ในการบ่ง precedence ในลำดับต่อไป แต่ก่อนอื่นเรามาพูดถึงการ scan expression เพื่อตรวจเช็คว่า token ที่ใช้ใน expression เป็น token ที่ใช้ได้

เราให้ว่า token ที่สามารถใช้ได้ใน expression ทางคณิตศาสตร์ของเราประกอบไปด้วย

- ตัวเลขฐาน 10 (decimal digit) ที่เป็นจำนวนเต็ม [0 | 1 | 2 | 3 |4 | 5 | 6 | 7 | 8 | 9]+
- operator ซึ่งประกอบไปด้วยสัญญลักษณ์ + - * / และ %
- วงเล็บเปิดและปิด (กับ)
- white space (ซึ่งไม่มีความสำคัญ และเราจะ scan ข้ามไป)

หน้าที่ของการตรวจจับ token ที่ถูกต้องจะเป็นหน้าที่ของ scanner ถ้ามีสัญญลักษณ์ใดๆที่แปลกปลอม นอกเหนือจาก token ที่ได้กล่าวมา scanner จะส่งข้อความเตือนว่ามี error เกิดขึ้น ซึ่งถ้าไม่ผ่านการ scan ก็จะไม่สามารถคำนวณหาผลลัพธ์ของ expression ได้ ตัวอย่างเช่น 23$5 ไม่ถือว่าเป็น token ที่ถูกต้องเพราะไม่อนุญาตให้มีสัญญลักษณ์ $ ใน token ใดๆ หลักการที่ scanner ใช้ในการตรวจจับ token ที่ถูกต้องก็คือการใช้ Regular Expression (RE) ซึ่งนิสิตได้เรียนรู้มาแล้วจากวิชา theory และเราจะได้พูดถึง RE อย่างละเอียดในเลคเชอร์ต่อๆไป

การมี token ที่ใช้ได้เพียงอย่างเดียวนั้น ไม่เพียงพอที่จะเป็น expression ทางคณิตศาสตร์ที่ถูกต้องได้ เราจำเป็นจะต้องมีโครงสร้างของ expression ที่ถูกต้องด้วย เช่น 76 + - * 3 ไม่ถือเป็น expression ที่ถูกต้อง แม้ว่า token ทุกตัวจะเป็น token ที่ใช้ได้ การกำหนดโครงสร้างของ expression ที่ถูกต้องและสามารถทำการคำนวณได้ เราจะใช้ context-free grammar (pushdown automata) เป็นตัวกำหนด grammar ด้านล่างนี้เป็น grammar ที่ถูกต้องสำหรับ expression ทางคณิตศาสตร์

```
expression = ["+" | "-"] , term , {("+" | "-") , term};
term       = factor , {("*" | "/" | "%") , factor};
factor     = constant | "(" , expression , ")";
constant   = digit , {digit};
digit      = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
```

ฟอร์มของ grammar ด้านบนเรียกว่า EBNF ซึ่งมีสัญลักษณ์ที่เกี่ยวข้องตามตารางดังต่อไปนี้

Usage	Notation
definition	=
concatenation	,
termination	;
alternation	|
option	[...]
repetition	{ ... }
grouping	(...)
terminal string	" ... "
terminal string	' ... '
comment	(* ... *)
special sequence	? ... ?
exception	-

ลองพิจารณาว่า expression ต่อไปนี้เป็น expression ที่ถูกต้องตาม grammar ด้านบนหรือไม่

33 + 437 * 9

ถูกต้อง โดยเราเริ่มพิจารณาว่า token (terminal) ทุกตัวเป็น token ที่ใช้ได้หมด จากนั้นพิจารณาจากกฎ 1. ไล่ลงไป ถ้าเราไม่คำนึงถึงสิ่งที่อยู่ใน {} เราจะได้ว่า expression คือ term คือ factor คือ constant ลองกระจายจาก non-terminal expression เรื่อยไปจะได้

```
expression = term + term
= factor + term
= constant + term
= constant + factor * factor
= constant + constant * constant
```

((33 + 437)) * 9

ถูกต้อง โดยเราเริ่มพิจารณาว่า token (terminal) ทุกตัวเป็น token ที่ใช้ได้หมด จากนั้นพิจารณาจากกฎ 1. ไล่ลงไป ถ้าเราไม่คำนึงถึงสิ่งที่อยู่ใน {} เราจะได้ว่า expression คือ term คือ factor คือ (expression) ลองกระจายจาก non-terminal expression เรื่อยไปจะได้

expression = term
= factor * factor
= (expression) * factor
= (term) * factor
= ((expression)) * factor
= ((term + term)) * factor
= ((factor + factor)) * factor
= ((constant + constant)) * constant

(1 *2+(3+4))+5

ถูกต้องโดยเราสามารถขยายกฎใน CFG จนกระทั่งได้ expression ด้านบนดังต่อไปนี้

expression = term + term
= factor + term
= (expression) + term
= (term + term) + term
= (factor * factor + term) + term
= (1 * factor + term) + term
= (1 * 2 + term) + term
= (1 * 2 + factor) + term
= (1 * 2 + (expression)) + term
= (1 * 2 + (term + term)) + term
= (1 * 2 + (factor + term)) + term
= (1 * 2 + (3 + term)) + term
= (1 * 2 + (3 + factor)) + T
= (1 * 2 + (3 + 4)) + T
= (1 * 2 + (3 + 4)) + factor
= (1 * 2 + (3 + 4)) + 5

4$3 % 998

ไม่ถูกต้อง 4$3 ไม่ใ.ช่ token ที่ใช้ได้

33 + -8 * 5

ไม่ถูกต้อง หลังจากเครื่องหมายบวกจะต้องเป็น term และจากกฎของ term ไม่มีกฎใดที่อนุญาตให้มีเครื่องหมาย - นำหน้า constant หรือ digit

ผลพลอยได้จากการระบุโครงสร้างของ expression ทางคณิตศาสตร์ด้วย grammar นี้คือเราได้ใส่ข้อมูลของ precedence ของ operator แต่ละตัวเข้าไปด้วย จะเห็นได้ว่า

- * / % มี precedence มากกว่า + -
- สิ่งที่อยู่ในวงเล็บมี precedence มากที่สุด

[ตัดไปที่ตัวโค๊ดของ parser ที่อาจารย์แจกให้ในชั่วโมงเรียน]

ลองเปรียบเทียบโค๊ดนี้กับ grammar ด้านบน จะเห็นได้ว่าโครงสร้างของโค๊ดกับตัว grammar มีโครงสร้างแบบเดียวกันนั่นคือ

- ฟังก์ชั่น SGet() ทำหน้าที่เป็นตัวประมวล token ที่ใช้ได้สำหรับ expression ทางคณิตศาสตร์ที่เราพิจารณา
- ตัวที่เป็น non-terminal (expression และ term และ factor) จะถูกทำเป็นฟังก์ชั่นและเรียกใช้งานในลักษณะ mutual recursion
- ในการเรียกใช้งานฟังก์ชั่นจะเป็นไปตามแบบแผนที่กำหนดโดย grammar

การ parse expression ในลักษณะนี้เราเรียกว่าการ parse แบบ recursive descent หรือ top-down parsing ซึ่งเป็นการ parse ที่เข้าใจได้ง่ายที่สุด และในวิชานี้เราจะยึดการ parse แบบนี้เป็นหลัก โดยการโค๊ด parser แบบนี้ทำได้อย่างตรงไปตรงมา เป็นการ parse แบบ top-down ตามตัว grammar โดยใช้ mutual recursion

หลังจากเราได้รับรู้ token ที่ใช้ได้ และโครงสร้างที่ถูกต้องของ expression ทางคณิตศาสตร์แล้ว ต่อไปเราจะมาคำนวณค่าผลลัพธ์ของ expression กัน โดยในขั้นตอนการหาผลลัพธ์นี้อาจารย์จะได้ให้เป็นการบ้าน ให้นิสิตได้เปลี่ยนแปลงโค๊ด parser ที่อาจารย์ให้ในชั้นเรียนเพื่อให้บรรลุวัตถุประสงค์ดังกล่าว