204324 ปฏิบัติการระบบคอมพิวเตอร์

ปฏิบัติการที่ 4: การ parse และประมวลผล expression

ปฏิบัติการช่วงที่เราจะได้ทำต่อไปนี้มีจุดประสงค์ให้นิสิตได้เรียนรู้กระบวนการคอมไพล์ expression เพื่อประมวลผล เราจะได้เห็นการนำความรู้ในวิชา theory of computing มาใช้ตั้งแต่ในขั้นตอนแรก เมื่อเราเข้าใจกระบวนการคอมไพล์ expression เราสามารถขยายแนวคิดนี้ต่อไปในการคอมไพล์โปรแกรมในภาษาระดับสูงได้

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 ถ้ามีสัญลักษณ์ใดๆที่แปลกปลอมนอกเหนือจาก 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

การมี token ที่ใช้ได้เพียงอย่างเดียวนั้น ไม่เพียงพอที่จะเป็น expression ทางคณิตศาสตร์ที่ถูกต้องได้ เราจำเป็นจะต้องมี โครงสร้างของ 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

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

ผลพลอยได้จากการระบุโครงสร้างของ expression ทางคณิตศาสตร์ด้วย grammar นี้คือเราได้ใส่ข้อมูลของ precedence ของ operator แต่ละตัวเข้าไปด้วย จะเห็นได้ว่า

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

ต่อไปนี้ขอให้พิจารณาโปรแกรม parser ที่แนบมากับปฏิบัติการนี้ (ไฟล์ expression_parser.pdf)

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 โดยการโค๊ด parser แบบนี้ทำได้อย่างตรงไปตรงมา เป็นการ parse แบบ top-down ตามตัว grammar โดยใช้ mutual recursion

หลังจากเราได้รับรู้ token ที่ใช้ได้ และโครงสร้างที่ถูกต้องของ expression ทางคณิตศาสตร์แล้ว ต่อไปเราจะมาประมวลผล expression กัน

ส่วนที่ 1

จากโค๊ด parser ที่ให้มา เขียนเพิ่มเติมเพื่อให้โค๊ดนี้ทำการคำนวณผลลัพธ์ของ expression ทางคณิตศาสตร์ได้ ตั้งชื่อไฟล์สำหรับโค๊ดใหม่นี้ว่า expreval.c ให้ส่งไฟล์นี้พร้อมกับกรณีทดสอบ 8 กรณีโดย 4 กรณีทำการคำนวณได้ถูกต้อง และอีก 4 กรณีมีโครงสร้างหรือ token ของ expression ไม่ถูกต้อง ทุกกรณีจะต้องมีการใช้ token ของ expression ครบทุกตัว

ส่วนที่ 2

จากโค๊ด parser ที่ให้มา ของ expression เขียนเพิ่มเติมเพื่อให้โค๊ดนี้ทำการสร้าง expression tree ตั้งชื่อโปรแกรมใหม่นี้ว่า parsetree.c เพื่อง่ายต่อการโปรแกรมในข้อนี้และข้ออื่นๆ ให้นิสิตตัดการมีเครื่องหมาย +/- หน้า term ออก นั่นคือเราจะไม่รองรับพวก expression ที่มี unary plus/minus

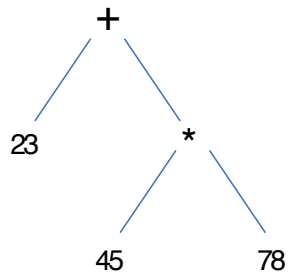
<u>การสร้าง expression tree</u>

จาก grammar สำหรับ expression ทางคณิตศาสตร์ เราสามารถที่จะใส่สิ่งที่ต้องกระทำ (semantic actions) หลังจากจบกฎ (production rule) ของ non-terminal โดยถ้า non-terminal เป็นฟังก์ชั่นในภาษาระดับสูง เราก็จะให้ฟังก์ชั่นนั้น return วัตถุ (object) หรือค่า (value) ที่ต้องการออกมา หลังจากได้เสร็จสิ้น semantic actions นั้นไปแล้ว

การหาผลลัพธ์ของ expression ทางคณิตศาสตร์เป็นตัวอย่างที่ง่ายที่สุดที่แต่ละ non-terminal ของ grammar จะ return ค่าที่เป็นจำนวนเต็ม ต่อไปเราจะมาดู semantic actions ในลักษณะที่แตกต่างไปบ้างนั่นคือการสร้าง expression tree ตัวอย่างเช่น expression ทางคณิตศาสตร์ต่อไปนี้

23 + 45 * 78

จะมี expression tree ที่เป็นตัวแทน expression นี้ดังต่อไปนี้



ในกรณีนี้เราจะให้ฟังก์ชั่นของ non-terminal return ค่า node ที่แทน node ใน expression tree (ถ้าจะให้ตรงจริงๆ สิ่งที่ return จะเป็น pointer ที่ชี้ไปที่ node) โดยนิยามของ node เป็นไปดังต่อไปนี้

```
typedef struct NodeDesc *Node;
typedef struct NodeDesc {
    char kind;          // plus, minus, times, divide, number
    int val;           // number: value
    Node left, right;  // plus, minus, times, divide: children
} NodeDesc;
```

ลองเปรียบเทียบฟังก์ชั่นที่แทน non-terminal ในกรณีที่เราต้องการคำนวณผลลัพธ์ของ expression กับกรณีที่เราต้องการจะสร้าง expression tree

```
static void int Factor(){
    int result;
    assert( (sym == number) || (sym == lparen) );
    if( sym == number ) {
        sym = SGet();
        return value;
    } else {
        sym = SGet();
        result = Expr();
        assert( sym == rparen );
        sym = SGet();
        return result;
    }
}
```

โค๊ดด้านบนแสดงการเปลี่ยนแปลงโค๊ดของการ parse expression สำหรับ non-terminal Factor เพื่อให้ทำการคำนวณผลลัพธ์ของ expression เปรียบเทียบกับโค๊ดด้านล่างของ Factor ที่ต้องการสร้าง expression tree

```
static Node Factor() {
     register Node result;
     assert( (sym == number) || (sym == lparen) );
     if( sym == number ) {
          result = malloc(sizeof(NodeDesc));
          result->kind  = number;
          result->val   = val;
          result->left  = NULL;
          result->right = NULL;
          sym = SGet();
     } else {
          sym = SGet();
          result = Expr();
          assert( sym == rparen );
          sym = SGet();
     }
     return result;
}
```

ในการแสดงผลของ expression tree ที่สร้างขึ้น ใช้ฟังก์ชั่น Print ดังต่อไปนี้ได้ ลองศึกษาฟังก์ชั่นนี้ดูและพยายามอธิบายว่าการทำงานของมันเป็นอย่างไร

```
static void Print( Node root, int level ) {
     register int i;

     if( root != NULL ) {
          Print( root->right, level+1 );
          for( i = 0; i < level; i++ ) printf(" ");
          switch( root->kind ) {
               case plus   : printf("+\n"); break;
               case minus  : printf("-\n"); break;
```

```
            case times  : printf("*\n"); break;
            case divide : printf("/\n"); break;
            case number : printf("%ld\n", root->val); break;
        }
        Print( root->left, level+1 );
    }
}
```

นำกรณีทดสอบทั้ง 4 ที่ถูกต้องจากในส่วนที่ 1 มาทดสอบการสร้าง expression tree และเรียกใช้งานฟังก์ชั่น Print เพื่อตรวจสอบดูว่าทุกอย่างถูกต้อง ส่งตัวอย่างหน้าจอที่พิมพ์ expression tree ด้วยฟังก์ชั่น Print ที่ถูกต้องมาหนึ่งภาพ

ส่วนที่ 3

เขียนฟังก์ชั่น Prefix Infix และ Postfix สามฟังก์ชั่น ที่พิมพ์ expression ที่รับเข้ามาเริ่มต้นในรูป prefix infix และ postfix ตามลำดับ โดยให้ expression แต่ละรูปแบบทั้งสามนั้นถูกพิมพ์อยู่ในบรรทัดเดียวกัน และมีช่องว่างหนึ่งช่องว่างระหว่าง token ของ expression ในการพิมพ์รูปแบบ infix นั้น นิสิตจะต้องใส่วงเล็บตาม precedence ให้ถูกต้อง เพื่อให้การคำนวณเป็นไปตาม expression เริ่มต้นที่ใส่เข้ามา สำหรับรูป prefix และ postfix นั้น ไม่ต้องมีการใส่วงล็บใดๆ precedence ของ operation ได้ถูกบ่งไว้ภายในรูปแบบของมันอยู่แล้ว เมื่อเขียนฟังก์ชั่นทั้งสามเรียบร้อยและทดสอบจนถูกต้องแล้ว ให้เซฟโปรแกรมที่มีการเพิ่มเติมใหม่นี้ลงในไฟล์ชื่อ preinpost.c

ให้เพิ่มเติมโค๊ด parser (และฟังก์ชั่น Print) ของเรา ให้สามารถรับ token ที่เป็นตัวอักษรเช่น x เพื่อจะนำมาใช้เป็นตัวแปรได้ โดยอาจจะให้ kind ของ node ที่บรรจุ x นี้เป็นแบบ var จากนั้นเขียนฟังก์ชั่นเพื่อคำนวณหา derivative ของ expression เริ่มต้นเมื่อเทียบกับตัวแปร x โดย expression เริ่มต้นนี้เป็นฟังก์ชั่นของ x ไม่ควรทำลาย expression tree ที่ได้มาจากการ parse expression เริ่มต้น แต่ควรจะสร้าง tree ใหม่ที่เป็น tree ของ expression ที่ได้มาจากการทำ derivative เมื่อเทียบกับตัวแปร x กับ expression เริ่มต้น ให้เซฟโปรแกรมที่มีการเพิ่มเติมใหม่นี้ลงในไฟล์ชื่อ diff.c

<u>ปฏิบัติการบน expression tree ตัวอย่างการหา derivative</u>

เมื่อเรามี expression tree เราสามารถทำ operation ได้หลายอย่างบน tree ที่เราสร้างขึ้น หนึ่งใน operation ที่เราจะมาพูดถึงกันคือการหา derivative ของ expression ที่มี token ที่เป็นตัวแปร เช่น x เพิ่มขึ้นมา ก่อนอื่นเราไปดูตัวอย่างของกฎการหา derivative เทียบกับ x ของฟังก์ชั่น f(x) และ g(x) ดังต่อไปนี้

- (f + g)' = f' + g'
- (f - g)' = f' - g'
- (f*g)' = f'*g + f*g'

จะเห็นได้ว่ากฎการ derivative เหล่านี้มีการนิยามในลักษณะ recursive โดยที่ termination condition ก็คือ

- (constant)' = 0
- ในกรณี f(x) = x จะได้ f' = 1

ดังนั้นถ้าเรามี expression tree ของ expression เริ่มต้น เราสามารถสร้าง tree ที่เป็น expression ที่เกิดการหา derivative ของ expression เริ่มต้นได้ไม่ยาก ขอให้นิสิตลองศึกษาแนวคิดจากโค๊ดต่อไปนี้

```
static Node diff ( Node root ){
  Node result;

  if ((root->kind == number) || (root->kind == var)) {
    create new "result" node
    if root->kind is number set result->value to 0
    else set result->value to 1
    set result->left and result->right to NULL;
     return result;
  }
  else if ((root->kind == plus) || (root->kind == minus)) {
    create new "result" node
    set root->kind to plus or minus accordingly
    set result->left to diff(root->left)
    set result->right to diff(root->right);
    return result;
  }
// more code to follow
```

ส่วนที่ 4

จากการทำ differentiation ของ expression เราจะได้ expression ผลลัพธ์ที่เยิ่นเย้อและสามารถที่จะลดรูป (simplify) ได้พอสมควร ตัวอย่างเช่น 0 + f หรือ 0 * f โดยที่ f เป็น subtree ของ expression ให้นิสิตเขียนโค๊ดเพื่อทำการ simplify ผลลัพธ์ของ differentiation โดยใช้กฎเกณฑ์ต่อไปนี้

0*f และ f*0 simplify เป็น 0
0+f และ f+0 simplify เป็น f
1*f และ f*1 simplify เป็น f
f-f simplify เป็น 0
f+f simplify เป็น 2*f

ให้นิสิตรวมโค๊ดใหม่ที่เพิ่มการ simplify ผลลัพธ์เข้าไปกับโค๊ดเดิมที่เขียนเพื่อทำ differentation แล้วตั้งชื่อไฟล์ใหม่ว่า simplify.c ซึ่งจะให้เอาพุทท์เพิ่มขึ้นมาต่อไปนี้คือ

- พิมพ์ผลลัพธ์ของการ differentiation ที่ simplify แล้วในรูป linear form
- พิมพ์ผลลัพธ์ของการ differentiation ที่ simplify แล้วในรูป tree form

สร้างกรณีทดสอบ 8 กรณีโดยที่มี 4 กรณีพื้นฐานที่ทดสอบ operator แต่ละตัวเช่น + - * / แบบโดดๆ และอีก 4 กรณีที่ซับซ้อน มีการใช้งาน operator ทั้ง 4 ตัวจนครบ ส่งกรณีทดสอบมาพร้อมกับโปรแกรมที่สมบูรณ์ และส่งภาพหน้าจอที่แสดงผลลัพธ์การประมวลผลกรณีทดสอบพื้นฐานและกรณีซับซ้อนมากรณีละหนึ่งภาพ

สิ่งที่ต้องส่งในชั่วโมง

- ส่วนที่ 1 และส่วนที่ 2
- ไฟล์ README เพื่อระบุสถานะว่าทำเสร็จถึงส่วนใด กรณีที่ทำไม่เสร็จให้ชี้แจงเหตุผลด้วย

สิ่งที่ต้องสิ่งหลังชั่วโมงปฏิบัติการ

- ส่วนที่ 3 และส่วนที่ 4
- ไฟล์ README เพื่อระบุสถานะว่าทำเสร็จถึงส่วนใด กรณีที่ทำไม่เสร็จให้ชี้แจงเหตุผลด้วย

การส่งงาน:

- **นำงานที่ต้องส่งในชั่วโมงปฏิบัติการใส่ไว้ในโฟลเดอร์ชื่อ studentID1_firstname1_studentID2_firstname2_lab4_part1 โดย studentID และ firstname คือ เลขประจำตัวและชื่อแรกของสมาชิกที่ทำปฏิบัติการร่วมกัน จากนั้น zip โฟลเดอร์นี้แล้วส่ง zip ไฟล์มาที่ Google Classroom ของวิชา *ภายในชั่วโมงปฏิบัติการ***
- **นำงานที่ต้องส่งหลังจากชั่วโมงปฏิบัติการใส่ไว้ที่โฟลเดอร์ studentID1_firstname1_studentID2_firstname2_lab4_part2 โดย studentID และ firstname คือ เลขประจำตัวและชื่อแรกของสมาชิกที่ทำปฏิบัติการร่วมกัน จากนั้น zip โฟลเดอร์นี้แล้วส่ง zip ไฟล์มาที่ Google Classroom ก่อนกำหนดส่ง**