204433 วิชาการแปลภาษาโปรแกรม

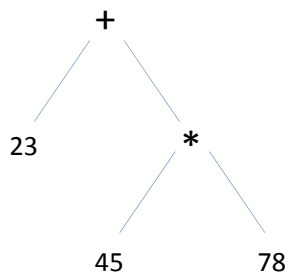
การสร้าง expression tree

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**) โดยนิยามขอบ **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)); // create new node for number
        result->kind  = number;
        result->val   = val;
        result->left  = NULL;
        result->right = NULL;
        sym = SGet();
    } else {
        sym = SGet();
        result = Expr(); // calls Expr() to build subtree
        assert( sym == rparen );
        sym = SGet();
    }
    return result;
}
```

เมื่อเราสร้าง expression tree ได้แล้ว เราสามารถจะ traverse tree นี้ได้ในรูปแบบทั้ง pre-order หรือ post-order หรือ in-order ได้ โดยถ้าเราพิมพ์ค่าของ node ขณะที่เรา visit ในแต่ละรูปแบบที่กล่าวมาแล้ว เราก็จะได้ expression ในรูป prefix หรือ postfix หรือ infix ตามลำดับ โดยการสร้าง expression tree และการพิมพ์ค่า expression ในรูปแบบต่างๆนี้ นิสิตจะได้ฝึกฝนในการบ้านที่ 2

ในการแสดงผลของ expression tree ที่สร้างขึ้น นิสิตสามารถใช้ฟังก์ชั่น 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 );
    }
}
```

ปฏิบัติการบน expression **tree** ***ตัวอย่างการหา*** *derivative*

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
```

Lexical Analysis (การตรวจจับคำศัพท์ในภาษาโปรแกรม)

ขั้นตอนแรกในการแปลภาษาระดับสูงคือการรู้จำ token และ non-token ที่ใช้ในภาษาระดับสูงนั้น เรียกขั้นตอนนี้ว่าการทำ lexical analysis ในโปรแกรมจะต้องมีเพียง token และ non-token ที่ภาษาอนุญาตให้ใช้เท่านั้น ในการระบุ token และ non-token ที่ภาษาอนุญาตให้ใช้จะกระทำโดยใช้ RE ซึ่งสามารถแทนได้โดยใช้ DFA หรือ NFA token ในภาษาโปรแกรมที่เราคุ้นเคยเช่น keyword ต่างๆ (if else while ฯลฯ) identifier ที่บ่งบอกถึงชื่อตัวแปร number ที่บ่งค่าของตัวแปร เป็นต้น พวก non-token เช่น comment หรือ directive ต่างๆ (เช่น #include) ได้อีกด้วย

ในการระบุ (specify) token หรือ non-token ที่ใช้ได้ในโปรแกรม เราจะใช้ RE ซึ่งมีความกระชับและชัดเจน หากแต่ RE ไม่ใช่รูปแบบที่เหมาะสมนักเวลาที่เราจะเขียนโค๊ด scanner เพื่อจะตรวจจับ token และ non-token รูปแบบการแทนที่เหมาะสมกว่าคือ DFA ซึ่งมีลักษณะเป็น state machine ที่ทำให้เราสามารถแทนด้วยโปรแกรมใน style แบบ table-driven ได้

เราจะอธิบายกระบวการทำ lexical analysis โดยเริ่มต้นจาก RE โดยเราจะให้ token แต่ละตัวถูกระบุโดย RE ดังนั้นถ้ามี token จำนวน n ตัวที่ภาษาโปรแกรมต้องการตรวจจับ และให้ว่า token แต่ละตัวแทนด้วย RE คือ R1 … Rn เราจะสามารถใช้ RE ในการบ่งบอก token ที่ภาษานี้รับได้คือ R1 | R2 | R3 | … | Rn จากนั้นเราจะแปลงจาก RE เป็น NFA และจาก NFA เป็น DFA ตามลำดับ จากวิชา theory ที่นิสิตได้ผ่านมา รูปแบบทั้งสามมีความสามารถเท่าเทียมกัน ไม่มีใครเด่นหรือด้อยไปกว่าใคร

DFA

DFA เป็น abstract machine (เครื่องยนต์เสมือน) ที่:

- อ่านอินพุท stream ของสัญญลักษณ์ (symbol) x $\in \Sigma$ โดย Σ เป็นพยัญชนะ (alphabet) ของ DFA
- มีจำนวน state จำกัด
- มี state เริ่มต้นที่ DFA machine เริ่มการอ่าน symbol
- ขณะที่มีการอ่าน symbol แต่ละอัน DFA จะมีการเปลี่ยน state โดยใช้ transition function δ นั่นคือเมื่อมีการอ่าน x เข้ามาขณะอยู่ที่ state q เราจะได้ว่า DFA จเปลี่ยน state ไปเป็น $q' = \delta(q, x)$
- มีเซ็ทของ state มากกว่าหรือเท่ากับหนึ่ง state ที่เป็น accept states เรียกเซ็ทนี้ว่า F เราจะได้ว่า DFA รู้จำ (accepts) อินพุท stream นี้ เมื่อสิ้นสุด stream แล้ว DFA อยู่ใน state q $\in$ F

NFA vs DFA

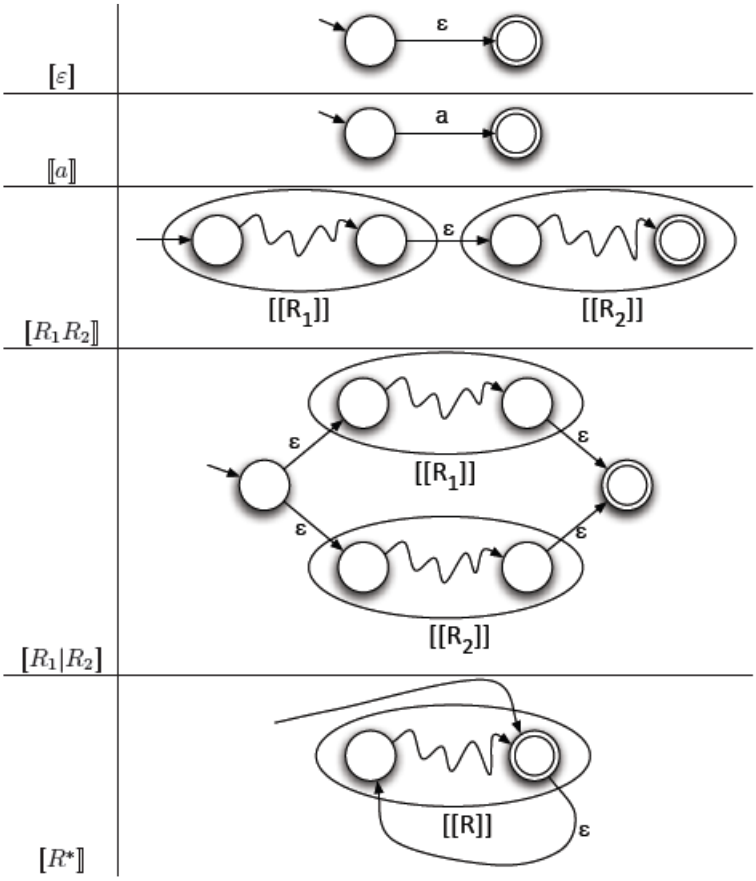
ความแตกต่างของ NFA จาก DFA คือการที่ NFA สามารถที่จะเปลี่ยนไปที่ state ได้มากกว่าหนึ่ง state เมื่อ

- อ่าน symbol แต่ละตัวใดๆ เข้ามา หรือ
- ไม่ได้อ่าน symbol ใดๆเข้ามา

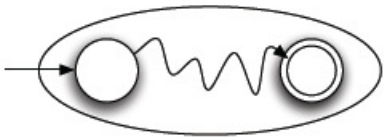
edge ที่บ่งการเปลี่ยน state โดยไม่มีการอ่านอินพุท เราจะใช้สัญญลักษณ์ ε label ไปที่ edge นั้น

แปลง RE ให้เป็น NFA

เราใช้หลักการ induction ถ้าเราให้ [[R]] แทนการแปลงจาก RE คือ R ให้เป็น NFA ซึ่งรับรู้ R เรานิยามปริมาณต่อไปนี้โดยแผนภาพ NFA ตามด้านล่าง

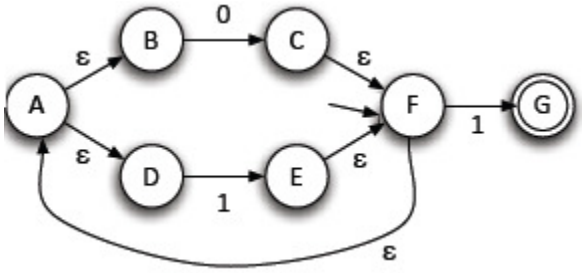


โดยแผนภาพ:



แทนการแปลง RE ใดๆเป็น NFA ที่มีหนึ่ง accept state

ถ้าเราต้องการรู้จำ token ที่เป็นจำนวนคี่ในระบบฐานสอง เราใช้ RE ต่อไปนี้ (0 | 1)*1 เราจะลองเปลี่ยน RE นี้ให้เป็น NFA ดังต่อไปนี้

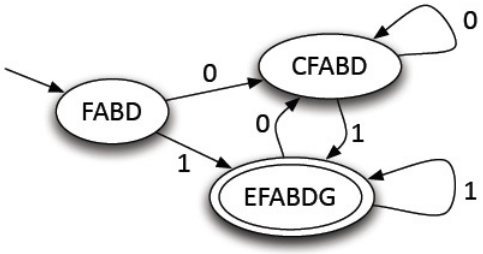


แปลงจาก NFA เป็น DFA

แนวทางการแปลงจาก NFA ให้เป็น DFA ทำได้ตามขั้นตอนดังต่อไปนี้

1. จาก state เริ่มต้น ดูว่าเราสามารถตาม edge Ɛ ไปได้ถึง state ใดๆบ้าง รวมกลุ่ม state เริ่มต้นกับ state ที่ไปถึงได้นี้ (เรียกการกระทำนี้ว่าการหา Ɛ-closure) เราได้ state แรกใน DFA ที่เราต้องการสร้าง
2. เลือก state ใดๆ จาก DFA ที่สร้างขึ้น และเลือก alphabet ใดๆ ที่ state ที่เลือก ยังขาด transition ที่ label ด้วย alphabet นี้ เติม edge นี้ลงใน DFA ที่ transition จาก state ที่เลือกไปที่ state ใหม่
3. state ใหม่จาก 2. จะรวมกลุ่ม state ใน NFA ที่สามารถไปถึงได้จาก transition ที่ label ด้วย alphabet ที่เลือกใน 2. บวกกับ Ɛ-closure ของกลุ่มของ state เหล่านี้ ถ้า state ใหม่นี้ซ้ำกับ state ของ DFA ที่จะสร้างขึ้น ให้ลบ state ใหม่นี้ทิ้งและให้ transition ที่ label ด้วย alphabet ใน 2.ไปที่ state เดิมที่ซ้ำกับ state ใหม่นี้แทน
4. ทำซ้ำขั้นตอนที่ 2. และ 3. จนกระทั่งทุก state ใน DFA ที่สร้างขึ้นครอบคลุมทุกๆ alphabet ที่จะมีการ transition ออกจาก state นั้น
5. กำหนด accept state ใน DFA ที่สร้างขึ้น โดยดูว่า state ใดๆใน DFA ที่เมื่อรวมกลุ่ม state ใน NFA มาแล้ว มี accept state ของ NFA เริ่มต้นอยู่ กำหนดให้ state ใน DFA นั้นเป็น accept state

เราสามารถ NFA ด้านบนที่แทน RE (0 | 1)*1 ให้เป็น DFA ได้ดังนี้



การลดรูป DFA ที่ได้ (DFA Minimization)

การสร้าง DFA จาก NFA ด้วยวิธีที่ได้กล่าวมา อาจจะมี state มากกว่าที่จำเป็น ต่อไปนี้เราจะมากล่าวถึงการตัด state ที่ "เกินมา" เหล่านั้นออกไปโดยใช้อัลกอริทึมของ John Hopcroft เราจะกล่าวถึงกฎที่บ่งบอกว่า state 2 state เท่ากันและสามารถลดรูป (merge) รวมกันได้ดังต่อไปนี้

กฎข้อที่ 1: state สอง state ใดๆ ไม่สามารถจะลดรูป (merge) รวมกันได้ ถ้าอันหนึ่งเป็น accept state แต่อีกอันหนึ่งไม่ใช่ accept state

กฎข้อที่ 2: ถ้า state 2 state q1 และ q2 ใดๆ transition จาก symbol x ใดๆ ไปที่ state ที่แตกต่างกัน จะได้ว่า q1 และ q2 แตกต่างกัน ไม่สามารถ merge รวมกันได้

อัลกอริทึมในการ miinimize DFA เป็นดังต่อไปนี้:

เริ่มต้นด้วยการสร้างตารางให้ชื่อว่า Distinct และให้ทุกๆช่องในตารางนี้ว่างเปล่า

(1) สำหรับ state 2 state (p, q) ใดๆ

If p is final and q is not, or vice versa,

Set Distinct(p, q) to be ϵ.

(2) วนลูปจนกระทั่งค่าทุกๆช่องในตาราง Distinct ไม่เปลี่ยนแปลง

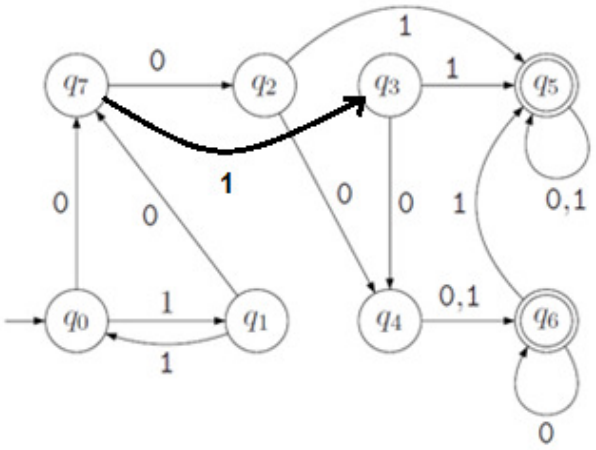
For each pair of states (p, q) and each character a in the alphabet:

if Distinct(p, q) is empty and Distinct$(\delta(p, \mathtt{a}),\ \delta(q, \mathtt{a}))$ is not empty

Set Distinct(p, q) to be a.

(3) state q และ q .ใดๆ ไม่สามารถ merge กันได้ถ้า ช่องในตาราง Distinct(p, q) ไม่ได้เป็นช่องว่างเปล่า

<u>ตัวอย่าง</u>



จาก DFA ด้านบน เราสร้างตาราง Distinct ตามอัลกอริทึมขั้นที่ (1) ได้ต่อไปนี้

q_0								
q_1								
q_2								
q_3								
q_4								
q_5	ϵ	ϵ	ϵ	ϵ	ϵ			
q_6	ϵ	ϵ	ϵ	ϵ	ϵ			
q_7						ϵ	ϵ	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

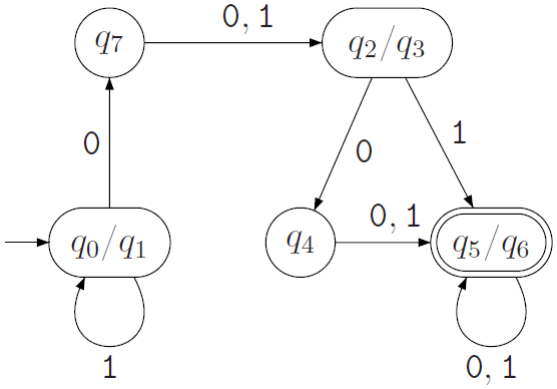
จะเห็นได้ว่าสำหรับ state 2 state (qi, qj) ใดๆ เรามีเพียง 1 ช่องในตาราง Distinct เพราะช่องสำหรับ (qj, qi) จะเหมือนกัน และเราไม่ต้องพิจารณาช่องสำหรับ (qi, qi)

และหลังจาก iteration ที่หนึ่งและสองในขั้นตอนที่ (2) ของอัลกอริทึม เราได้ตาราง Distinct ดังต่อไปนี้

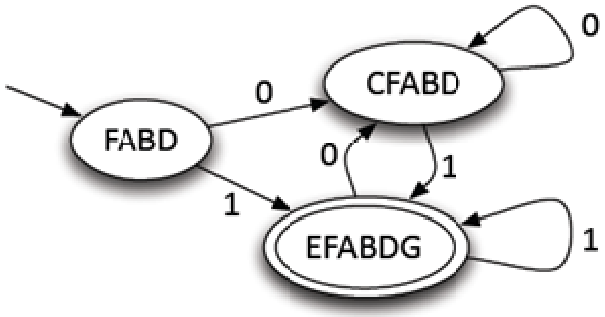
q_0								
q_1								
q_2	1	1						
q_3	1	1						
q_4	0	0	0	0				
q_5	ϵ	ϵ	ϵ	ϵ	ϵ			
q_6	ϵ	ϵ	ϵ	ϵ	ϵ			
q_7			1	1	0	ϵ	ϵ	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

q_0								
q_1								
q_2	1	1						
q_3	1	1						
q_4	0	0	0	0				
q_5	ϵ	ϵ	ϵ	ϵ	ϵ			
q_6	ϵ	ϵ	ϵ	ϵ	ϵ			
q_7	1	1	1	1	0	ϵ	ϵ	
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7

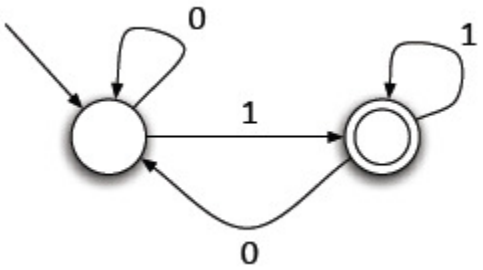
หลังจาก iteration ที่สอง ค่าใน Distinct ไม่มีการเปลี่ยนแปลง เราจะได้ว่า state (q0, q1) (q2, q3) และ q5, q6) สามารถ merge รวมกันได้ และเราจะได้ DFA ที่ minimize แล้วดังต่อไปนี้



กลับไปดูที่ตัวอย่าง DFA ที่แทน RE (0 | 1)*1 ที่เราได้จากการเปลี่ยนมาจาก NFA ในเลคเชอร์ที่แล้ว



นิสิตลอง minimize DFA ด้านบน ดูว่าได้ DFA ที่ minimize แล้วตามรูปด้านล่างนี้หรือไม่



Pseudo-code สำหรับตรวจจับ string (scanner) ที่ต้องการโดยใช้ DFA

เราสามารถแทน transition function ของ DFA ด้านบนได้โดยใช้ตาราง (table) ด้านล่างต่อไปนี้

	0	1
q_0	q_0	q_1
q_1	q_0	q_1

Pseudo-code สำหรับ DFA ด้านบน ที่รับอินพุทเป็น string ความยาว n และ input[i] คือ character ตัวที่ i ของ string ความยาว n นี้ สามารถเขียนได้ดังต่อไปนี้f

```
start := i
q := q0
while (i <= n) {
    q := δ(q, input[i])
    i := i + 1
}
if (q ϵ F) return accept
else return fail
```

Longest Match

เมื่อเราได้ DFA สำหรับการรู้จำ token ในภาษาโปรแกรมมาแล้ว เรายังมีสิ่งที่ต้องคำนึงเพิ่มเติม นั่นคือ DFA ที่ได้มานั้น อาจจะทำให้เกิด ambiguity (การที่มีความเป็นไปได้ในหลายๆทาง) ตัวอย่างเช่น ถ้าอ่าน อินพุท if8 เข้ามา เราจะคิดว่าอินพุทนี้แบ่งเป็น 2 token คือ if และ 8 หรือเป็น token เดียวคือ if8 เมื่อเป็นเช่นนี้เราจะใช้กฎ longest match ในการกำจัด ambiguity นี้ โดยให้ว่าเราจะอ่านอินพุทไปจนกว่าจะถึงจุดที่ DFA ไปต่อไม่ได้ แล้วพิจารณา accept state ครั้งล่าสุดที่ match ได้ พิจารณา scanner ด้านล่างนี้ ที่ implement ในฟังก์ชั่น NextWord() ที่ return token ที่รู้จำได้จาก input ที่เข้ามาต่อไปนี้ โดย s_e และ S_A แทน error state และ accept state ตามลำดับ

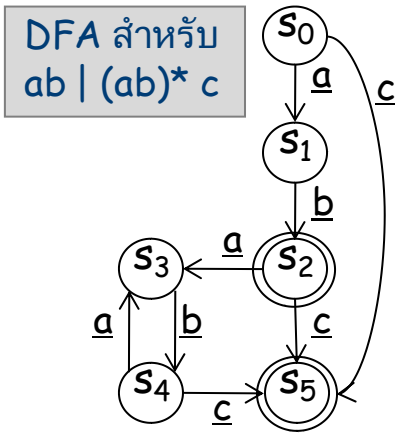
```
// recognize words
NextWord() {
    state ← s0
    lexeme ← empty string
    clear stack
    push (bad)
    while (state ≠ se) do
       char ← NextChar( )
       lexeme ← lexeme + char
       if state ∈ SA
          then clear stack
       push (state)
       state ← δ(state,char)
       end;

    // clean up final state
    while (state ∉ SA and state ≠ bad) do
        state ← pop()
        truncate lexeme
        roll back the input one character
        end;
    // report the results
    if (state ∈ SA )
        then return lexeme
        else  return invalid
}
```

การทำ longest match อาจจะทำให้เกิดสภาวะที่เราไม่ประสงค์ขึ้นมาได้ นั่นคือการรู้จำ token อาจจะใช้เวลา asymptotic ที่มากกว่า O(n) เป็น $O(n^2)$ โดย n คือความยาวของ token ที่ต้องการรู้จำ พิจารณาการรู้จำ token ที่มี RE ดังต่อไปนี้:

ab | (ab)*c

พิจารณา DFA ที่ใช้รู้จำ token นี้และโค๊ดด้านบน



พิจารณาถ้า input stream ที่เข้ามาเป็น abababababc ในการรู้จำ token นี้ ใช้เวลา linear time ตามโค๊ดด้านบน โดย loop while ที่สองที่ใช้ในการ rollback จะไม่ได้ทำงานเลย

ต่อไปลองพิจารณาว่าถ้า input stream ที่เข้ามาเป็น abababab โค๊ดด้านบนจะต้องอ่าน input stream ไปเรื่อยๆจนกระทั่งสิ้นสุด จึงพบว่า longest match จริงๆคือ ab นั่นคือเมื่อไปถึง b ตัวสุดท้าย แล้วหลุดจาก while loop แรกออกมา while loop ที่สองจะทำการ rollback จนกระทั่งมีการ pop state s2 ออกมา และได้ lexeme เป็น ab ในครั้งต่อไปที่ NextWord ถูกเรียก มันจะเริ่มต้นจาก a ตัวที่สอง และก็กลับมาทำกระบวนการเดิมที่ต้อง rollback อีก เช่นนี้ทำให้มี runtime เป็น $O(n^2)$

เราสามารถแก้ปัญหาของการ rollback ในทุกครั้งๆเพื่อตัด lexeme ab ให้ได้จาก input stream นี้โดยใช้โค๊ด scanner แบบ Maximum Munch โดยปรับปรุง Pseudo-code ของ NextWord ดังต่อไปนี้

```
// recognize words
NextWord() {
    state ← s0
    lexeme ← empty string
    clear stack
    push (bad,bad)
    while (state ≠ se) do
       char ← NextChar( )
       InputPos ← InputPos + 1
       lexeme ← lexeme + char
       if Failed[state,InputPos]
           then break;
       if state ∈ SA
          then clear stack
       push (state,InputPos)
       state ← δ(state,char)
       end

    // clean up final state
    while (state ∉ SA and state ≠ bad) do
        Failed[state,InputPos) ← true
       ⟨state,InputPos⟩ ← pop()
        truncate lexeme
        roll back the input one character
        end
    // report the results
    if (state ∈ SA )
        then return lexeme
        else  return invalid
}
```