Regular Expression

Course Code:CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:		**Week No:**		**Semester:**	
Lecturer:	*Md Masum Billah, billah.masumcu@aiub.edu*				

Lecture Outline

1. Definition of a Regular Expression
2. Rules of a Regular Expression
3. Examples
4. Exercises

Objectives and Outcomes

Objectives:

- Understand the basic concept of Regular expression
- Understand the regular expression algorithm

Outcome:

- Students should be able to design the nondeterministic finite automate from regular expression.
- Students should be able to know the applications of a regular expression.

Regular Expression

Definition: A sequence of symbols and characters expressing a string or pattern to be searched for within a longer piece of text.

Another words to say a regular expression is a method used in programming for pattern matching. Regular expressions provide a flexible and concise means to match strings of text.

The regular expressions are built recursively out of smaller regular expressions, using some rules.
Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r ' s subexpressions.

Regular Expression

The regular expressions are built recursively out of smaller regular expressions, using some rules.

Each regular expression r denotes a language L(r), which is also defined recursively from the languages denoted by r ' s subexpressions.

Rules of Regular Expression

Here are the rules that define the regular expressions over some alphabet £ and the languages that those expressions denote.

- Basis
- Induction
- Precedence

Rules of Regular Expression

BASIS: There are two rules that form the basis:

- E is a regular expression, and *L(E)* is {E}, that is, the language whose sole member is the empty string.

- If *a* is a symbol in E, then **a** is a regular expression, and L**(a) =** *{a},* that is, the language with one string, of length one, with *a* in its one position. Here italics is used for symbols, and boldface for their corresponding regular expression.

Rules of Regular Expression

INDUCTION: There are four parts to the induction. Suppose r and *s* are regular expressions denoting languages *L(r)* and *L(s),* respectively.

- (r)|(s) is a regular expression denoting the language *L(r)* U *L(s).*
- (r)(s) is a regular expression denoting the language *L(r)L(s).*
- (r)* is a regular expression denoting (L(r))*.
- (r) is a regular expression denoting *L(r).*The last rule says that we can add additional pairs of parentheses around expressions without changing the language they denote.

Example of a Regular expression

Let E = {a, b}.

- 1. The regular expression **a|b** denotes the language *{a, b}.*
- 2. **(a|b)(a|b)** denotes *{aa, ab, ba, bb},* the language of all strings of length two over the alphabet E.
- Another regular expression for the same language is **aa|ab|ba|bb.**
- 3. **a*** denotes the language consisting of all strings of zero or more a's, that is, { E, *a , a a , a a a , . . . }.*

Example of a Regular expression

Let E = {a, b}.

- 4. **(a|b)*** denotes the set of all strings consisting of zero or more instances of *a* or *b,* that is, all strings of a's and b's: {E *,a, b,aa, ab, ba, bb,aaa,...}.*

- Another regular expression for the same language is **(a*b*)*.**

- **a|a*b** denotes the language *{a, b, ab, aab, aaab,...*}, that is, the string *a* and all strings consisting of zero or more a's and ending in *b.*

Operations of a Regular expression

Operations:

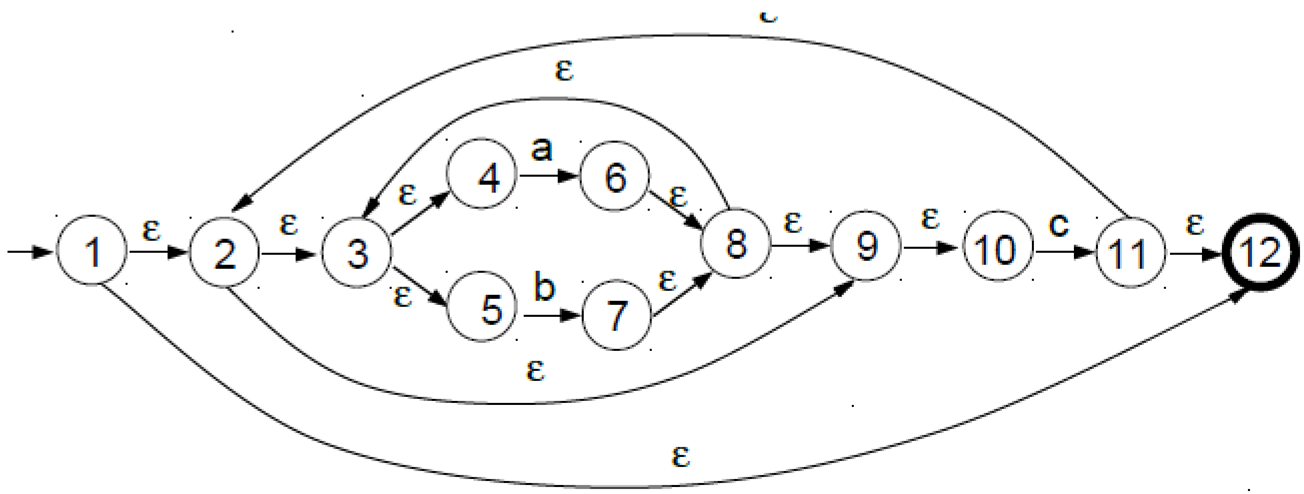
The various operations on languages are:

- Union of two languages L and M is written as
  L U M = {s | s is in L or s is in M}
- Concatenation of two languages L and M is written as
  LM = {st | s is in L and t is in M}
- The Kleene Closure of a language L is written as
  L* = Zero or more occurrence of language L.

Regular Expression To NFA

Outline the NFA generated by the construction of Thompson relevant to the following regular expression:

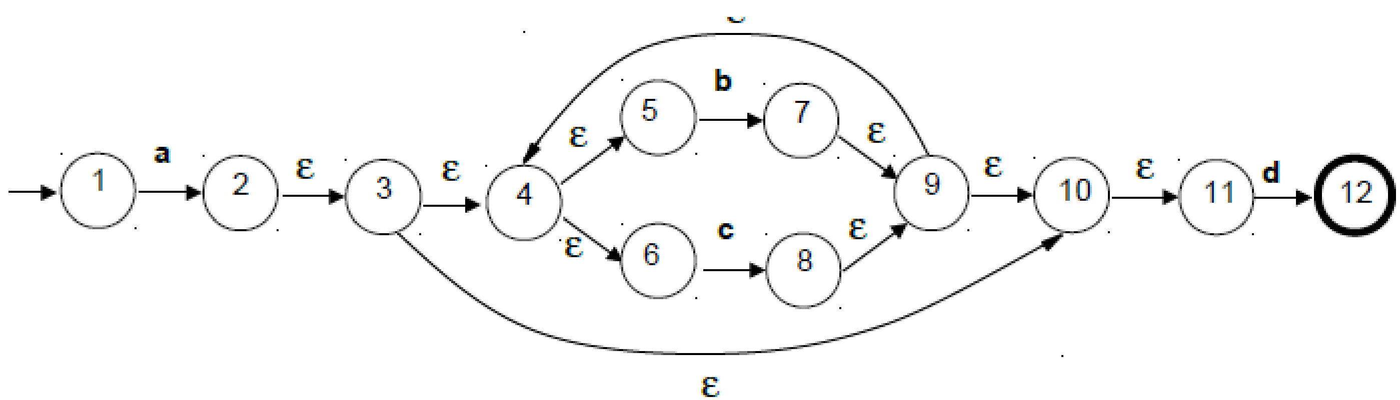
Example: ((a | b)*c)*



Regular Expression To NFA

By means of the construction of Thompson, outline the NFA relevant to the following regular expression:

Example: a (b | c)*d



Regular Expression To NFA

By means of the construction of Thompson, outline the NFA relevant to the following regular expression:

Example: a (b | c)*d

Class Exercises

1. (aUb)*abc

1. (abUbc(abUc)*)*

Lecture References

A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools*** *(*The Dragon Book*)*, [Second Edition]

References

1. A. Aho, R. Sethi and J. Ullman, ***Compilers: Principles, Techniques and Tools(***The Dragon Book*),* [Second Edition]

2. **Principles of Compiler Design** (2nd Revised Edition 2009) A. A. Puntambekar

3. Basics of Compiler Design Torben Mogensen

NFA to DFA Conversion (Subset Construction Method)

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	8	**Week No:**	8	**Semester:**	
Lecturer:	*Md Masum Billah, billah.masumcu@aiub.edu* *Masum-billah.net*				

Lecture Outline

1. NFA TO DFA (Subset Construction Method)
2. Subset Construction Algorithm
3. DFA Designing
4. Example
5. Exercise
6. References

Objective and Outcome

Objective:

- To explain the subset construction algorithm/method for converting a Non deterministic machine to deterministic machine.
- Provide necessary example and explanation of NFA to DFA conversion method using subset construction method.
- To explain and practice Deterministic Finite Automata (DFA) Machine Design for a given Grammar.

Outcome:

- After this lecture the students will be capable of demonstrating the subset construction algorithm
- After this lecture the student will be able to convert an NFA to relevant DFA by following subset construction method.
- After this class student will be able to design and demonstrate DFA construction from a given Grammar.

NFA to DFA Conversion

Subset Construction Algorithm

Input: An NFA N

Output: A DFA D accepting the same language

Method: Constructs a transition table Dtran for D. Each DFA state is a set of NFA states and construct Dtran so that D will simulate "in parallel" all possible moves N can make on a given input string

OPERATION	DESCRIPTION
$\epsilon\text{-}closure(s)$	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
$\epsilon\text{-}closure(T)$	Set of NFA states reachable from some NFA state s in T on ϵ-transitions alone.
$move(T, a)$	Set of NFA states to which there is a transition on input symbol a from some NFA state s in T.

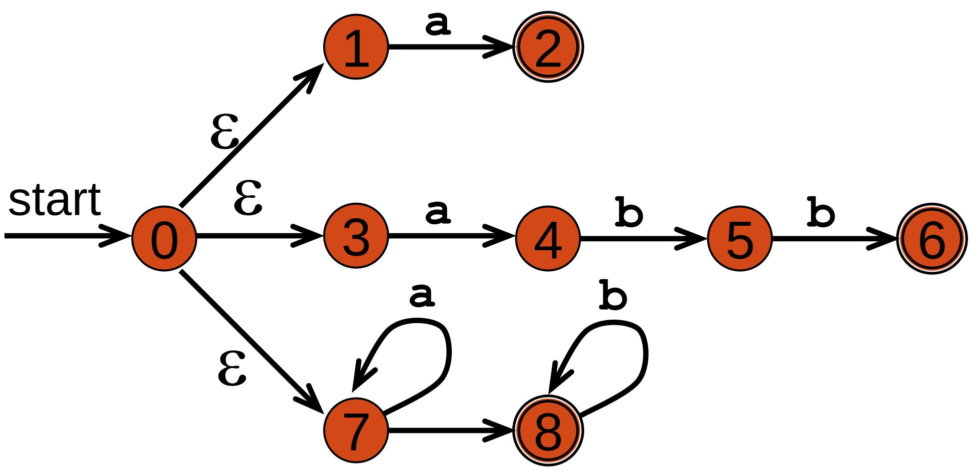
NFA to DFA Conversion

Subset Construction Algorithm

initially, ϵ-*closure*(s_0) is the only state in *Dstates* and it is unmarked;
while there is an unmarked state *T* in *Dstates* **do begin**
    mark *T*;
    for each input symbol *a* **do begin**
        U := ϵ-*closure*(*move*(*T*, *a*));
        if *U* is not in *Dstates* **then**
            add *U* as an unmarked state to *Dstates*;
        Dtran[*T*, *a*] := *U*
    end
end

NFA to DFA Conversion

ε-*closure* and *move* Examples



ε-*closure*({0}) = {0,1,3,7}

move({0,1,3,7},**a**) = {2,4,7}

ε-*closure*({2,4,7}) = {2,4,7}

move({2,4,7},**a**) = {7}

ε-*closure*({7}) = {7}

move({7},**b**) = {8}

ε-*closure*({8}) = {8}

move({8},**a**) = ∅

Alphabet / Symbol = {a, b}

Subset Construction Algorithm

Subset Construction Algorithm

The *subset construction algorithm* converts an NFA into a DFA using:

$$\varepsilon\text{-}closure(s) = \{s\} \cup \{t \mid s \to_\varepsilon \ldots \to_\varepsilon t\}$$
$$\varepsilon\text{-}closure(T) = \cup_{s \in T}\, \varepsilon\text{-}closure(s)$$
$$move(T,a) = \{t \mid s \to_a t \text{ and } s \in T\}$$

The algorithm produces:

- D_{states} is the set of states of the new DFA consisting of sets of states of the NFA
- D_{tran} is the transition table of the new DFA

Subset Construction Algorithm

Algorithm Explained

1. Create the start state of the DFA by taking the ε-closure of the start state of the NFA
2. Perform the following for the DFA state:
   - Apply move to the newly-created state and the input symbol; this will return a set of states.
   - Apply the ε-closure to this set of states, possibly resulting in a new set.

     This set of NFA states will be a single state in the DFA.
3. Each time we generate a new DFA state, we must apply step 2 to it. The process is complete when applying step 2 does not yield any new states.
4. The finish states of the DFA are those which contain any of the finish states of the NFA

Subset Construction Algorithm

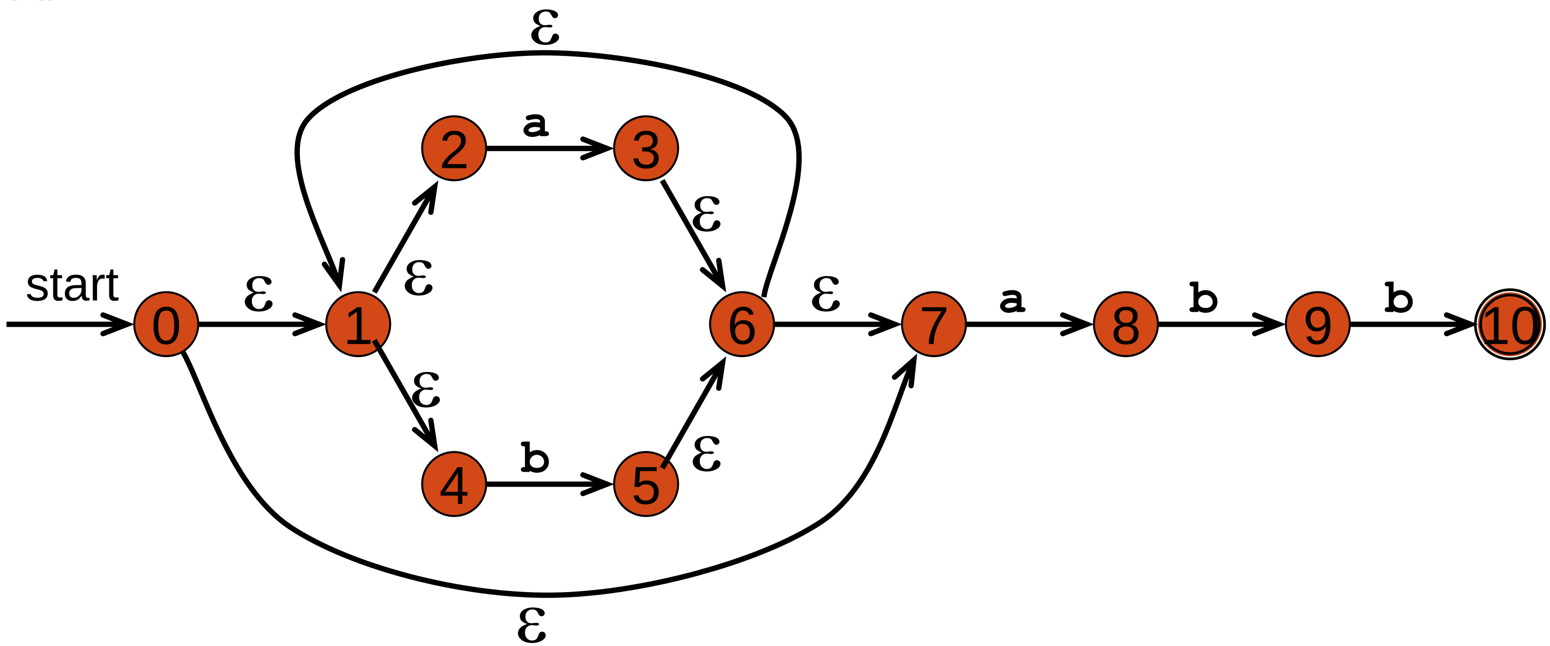
Algorithm with while Loop

```
fun nfa2dfa start edges =
 let val chars = nodup(sigma edges)
     val s0 = eclosure edges [start]
     val worklist = ref [s0]
     val work = ref []
     val old = ref []
     val newEdges = ref []
 in while (not (null (!worklist))) do
     ( work := hd(!worklist)
     ; old := (!work) :: (!old)
     ; worklist := tl(!worklist)
     ; let fun nextOn c = (Char.toString c
                    ,eclosure edges (nodesOnFromMany (Char c) (!work) edges))
          val possible = map nextOn chars
          fun add ((c,[])::xs) es = add xs es
            | add ((c,ss)::xs) es = add xs ((!work,c,ss)::es)
            | add [] es = es
          fun ok [] = false
            | ok xs = not(exists (fn ys => xs=ys) (!old)) andalso
                  not(exists (fn ys => xs=ys) (!worklist))
          val new = filter ok (map snd possible)
       in worklist := new @ (!worklist);
         newEdges := add possible (!newEdges)
       end
     );
    (s0,!old,!newEdges)
 end;
```

NFA to DFA Conversion

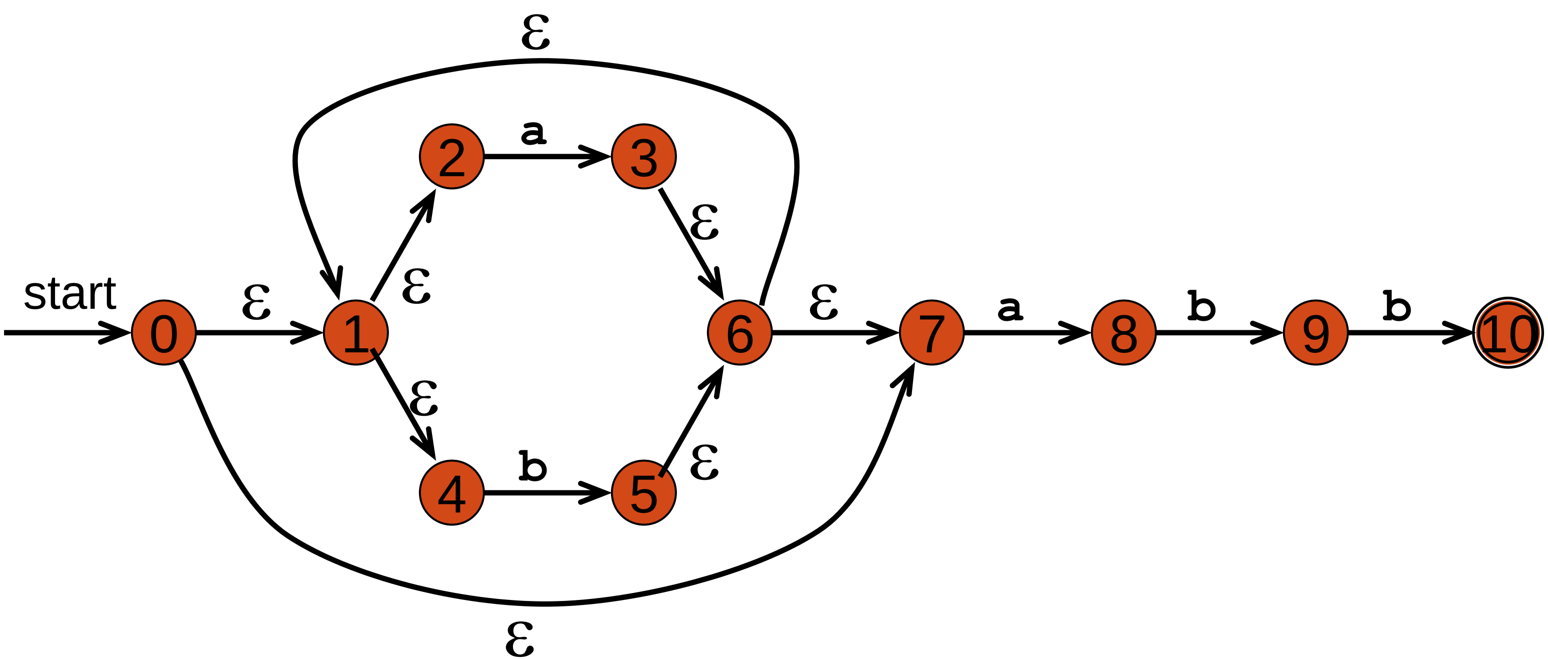
Subset Construction Method (Example-1)

NFA:



Regular Expression: (a | b)* abb

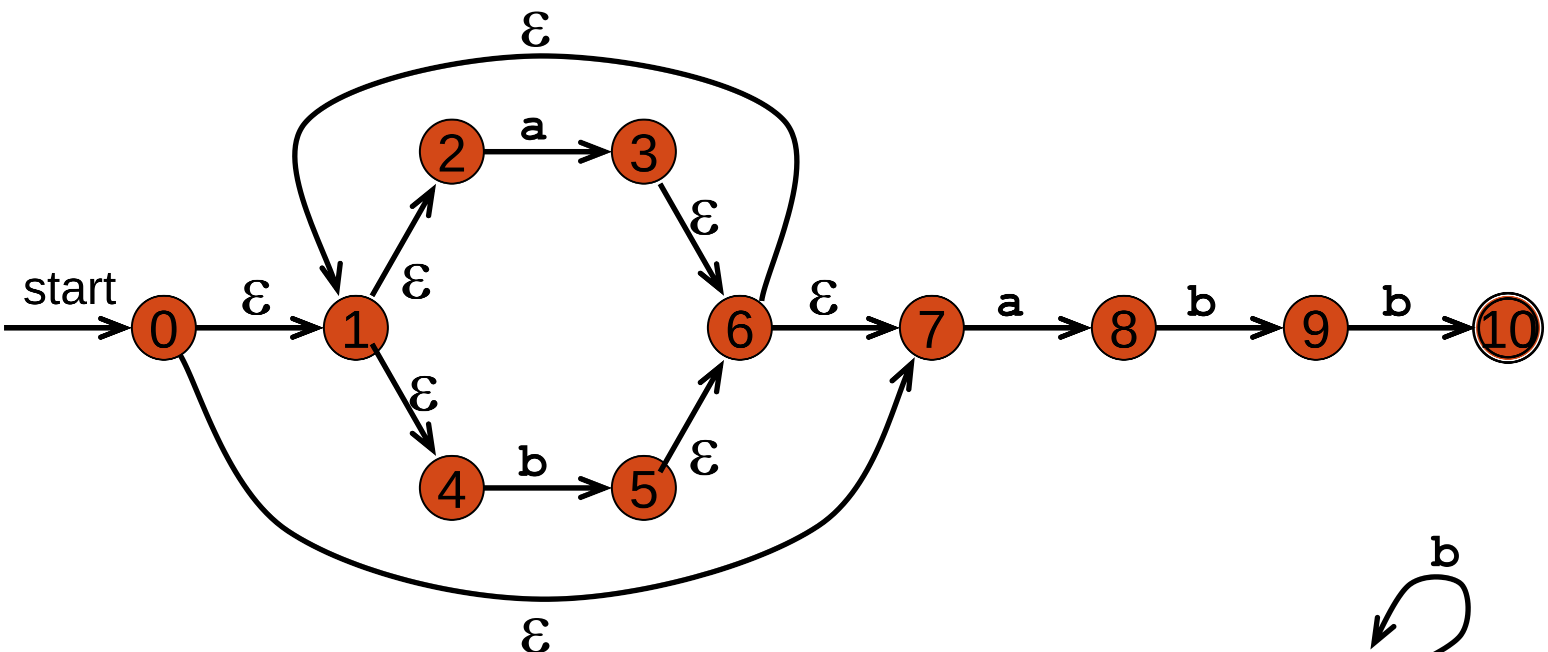
Subset Construction Method (Example-1)



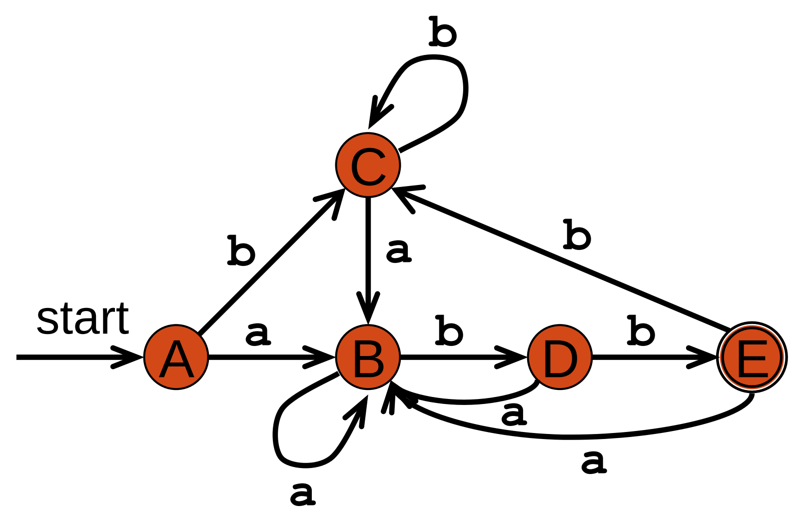
DFA State	E-closure of	E-closure outcome states
A	E-closure ({0})	0,1,2,4,7
B	E-closure ({3,8})	1,2,3,4,6,7,8
C	E-closure ({5})	1,2,4,5,6,7
D	E-closure({5,9})	1,2,4,5,6,7,9
E	E-closure({5,10})	1,2,4,5,6,7,10

NFA States	DFA State	a	b
0,1,2,4,7	A	B	C
1,2,3,4,6,7,8	B	B	D
1,2,4,5,6,7	C	B	C
1,2,4,5,6,7,9	D	B	E
1,2,4,5,6,7,10	E	B	C

Subset Construction Method (Example-1 Cont.)



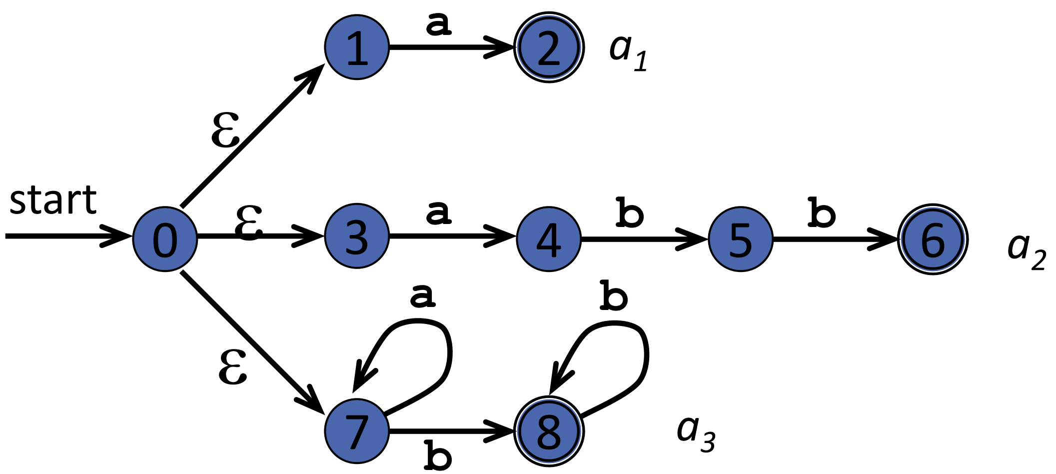
NFA State	DFA State	a	b
0,1,2,4,7	A	B	C
1,2,3,4,6,7,8	B	B	D
1,2,4,5,6,7	C	B	C
1,2,4,5,6,7,9	D	B	E
1,2,4,5,6,7,10	E	B	C



NFA to DFA Conversion

Subset Construction Method (Exercise 1)

NFA

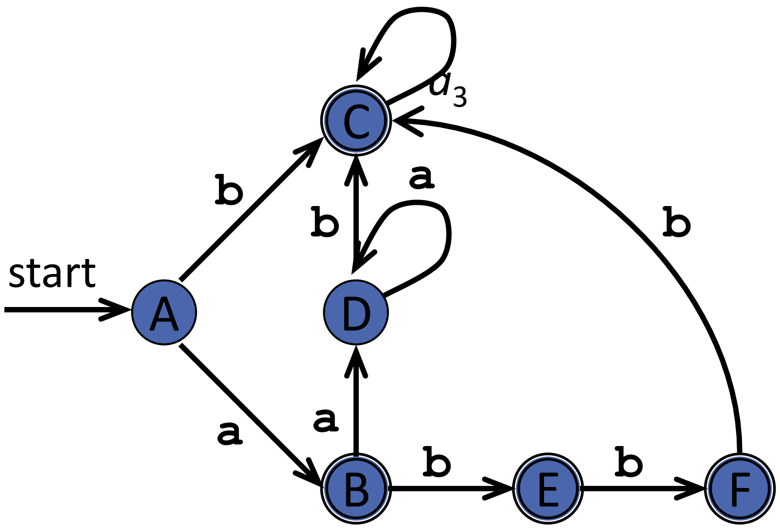


Converted DFA in the next Slide

NFA to DFA Conversion

Subset Construction Method (Exercise 1)

DFA



Dstates

A = {0,1,3,7}

B = {2,4,7}

C = {8}

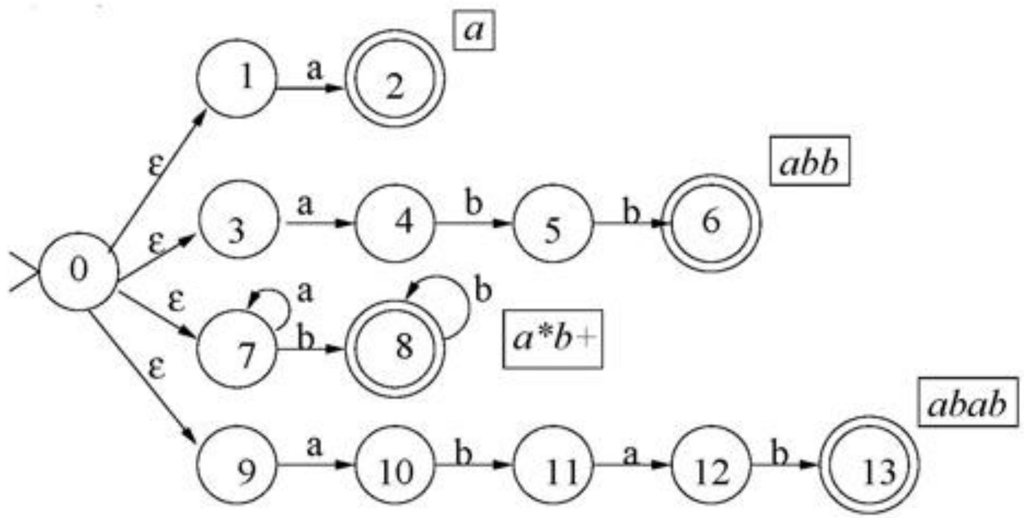
D = {7}

E = {5,8}

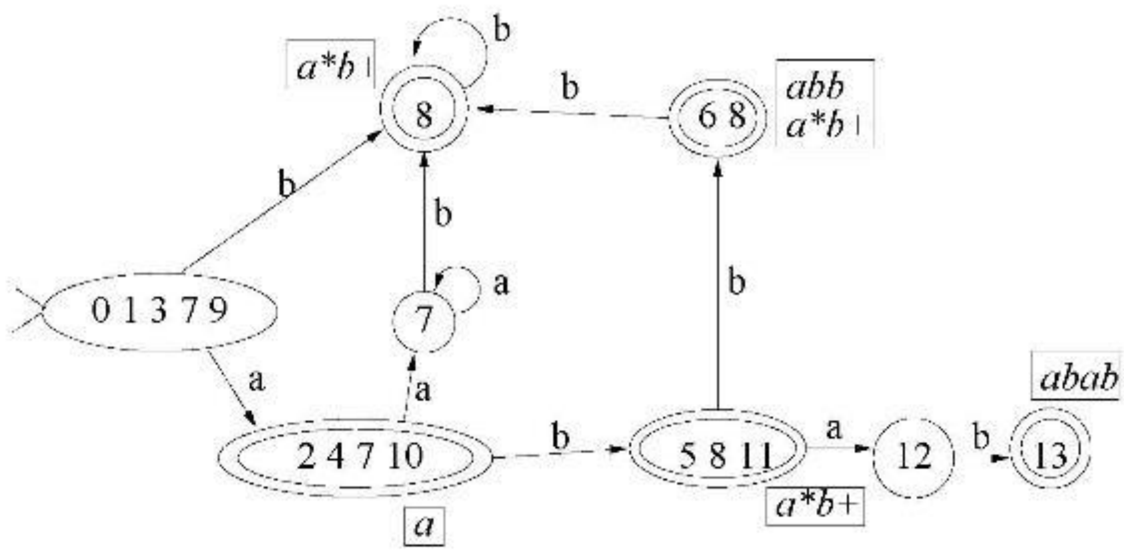
F = {6,8}

NFA to DFA / Subset Construction Method (Exercise 2)

NFA



DFA
Hints



Deterministic Finite Machine

DFA DESIGN

- A finite automaton is a 5-tuple (Q, Σ, δ, q_0, F), where
  - Q is a finite set called the ***states***,
  - Σ is a finite set called the ***alphabet***,
  - $\delta : Q \times \Sigma \rightarrow Q$ is the ***transition function***,
  - $q_0 \in Q$ is the ***start state***,
  - $F \subseteq Q$ is the set of ***accept*** (*final*) ***states***.
- If A is the set of all strings that a machine *M* accepts, we say that *A* is the ***language of machine M*** and write *L*(*M*)=*A*, ***M recognizes A*** or ***M accepts A***.

Deterministic Finite Machine

DFA Example 1

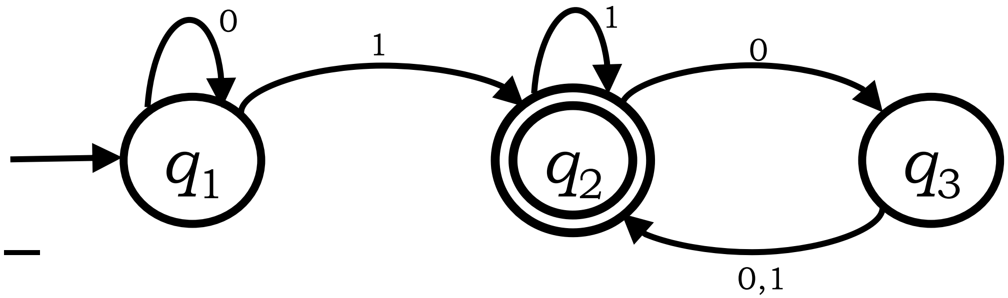


Figure: Finite Automaton M_1

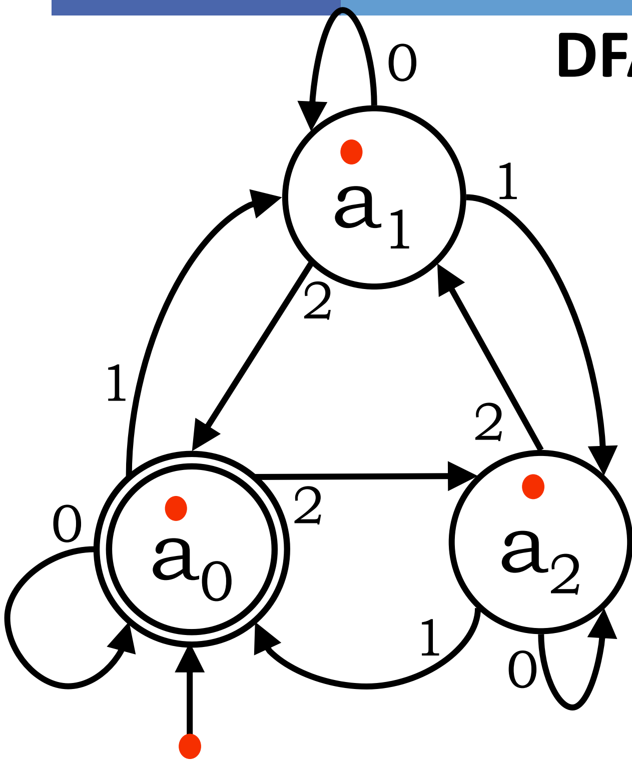
- $M_1 = (Q, \Sigma, \delta, q_0, F)$, where –
  - $Q = \{q_1, q_2, q_3\}$,
  - $\Sigma = \{0, 1\}$,
  - *δ is describe as –*
  - $q_0 = q_1$,
  - $F = \{q_2\}$.

δ	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

or

$$\delta(q_1,0) = q_1,\ \delta(q_1,1) = q_2,$$
$$\delta(q_2,0) = q_3,\ \delta(q_2,1) = q_2,$$
$$\delta(q_3,0) = q_2,\ \delta(q_3,1) = q_2.$$

DFA Design Example



- Input example: 01120101
- Present State:

  a_2

- Input symbol:

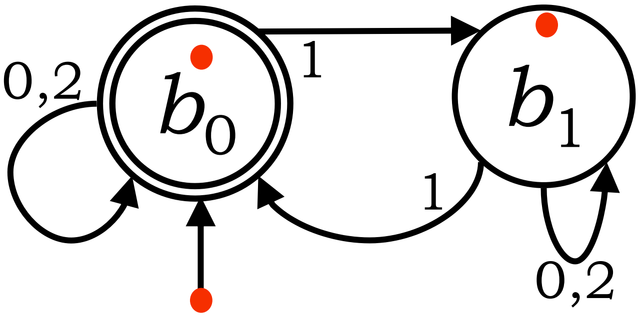
  ε

Accepted

- Alphabet Σ={0,1,2}.
- Language A_1 = {*w* : the sum of all the symbols in *w* is multiple of 3 }.
  - Can be represented as follows –
    - *S*= the sum of all the symbols in *w*.
    - If *S* modulo 3 = 0 then the sum is multiple of 3.
    - So the sum of all the symbols in *w* is 0 modulo 3.
    - Here, a_i is modeled as *S* modulo 3 = *i*.
- The finite state machine M_1= (Q_1, Σ, δ_1, q_1, F_1), where –
  - $Q_1 = \{a_0, a_1, a_2\}$,
  - $q_1 = a_0$,
  - $F_1 = \{a_0\}$,
  - δ_1

	0	1	2
a_0	a_0	a_1	a_2
a_1	a_1	a_2	a_0
a_2	a_2	a_0	a_1

DFA Design Example



- Alphabet $\Sigma=\{0,1,2\}$.
- Language $A_1 = \{w$: the sum of all the symbols in w is an even number $\}$.
  - Can be represented as follows –
    - S= the sum of all the symbols in w.
    - If S modulo 2 = 0 then the sum is even.
    - Here, b_i is modeled as S modulo 2 = i.
- The finite state machine $M_2= (Q_2, \Sigma, \delta_2, q_2, F_2)$, where –
  - $Q_2 = \{b_0,b_1\}$,
  - $q_2 = b_0$,
  - $F_2 = \{b_0\}$,
  - δ_2

	0	1	2
b_0	b_0	b_1	b_0
b_1	b_1	b_0	b_1

- Input example: 01120101
- Present State:

b_1

- Input symbol:

ε

Accepted

DFA Design Example (Type 1)

The construction of DFA for languages consisting of strings ending with a particular substring.

- Determine the minimum number of states required in the DFA.
  - Calculate the length of substring.
  - All strings ending with 'n' length substring will always require minimum (n+1) states in the DFA.
- Draw those states.
- Decide the strings for which DFA will be constructed.
- Construct a DFA for the decided strings
  - While constructing a DFA, Always prefer to use the existing path. Create a new path only when there exists no path to go with.
- Send all the left possible combinations to the starting state.
- Do not send the left possible combinations over the dead state.

DFA Design Example and Exercise

- Draw a DFA for the language accepting strings ending with ‘abb’ over input alphabets ∑ = {a, b}
- Draw a DFA for the language accepting strings starting with ‘ab’ over input alphabets ∑ = {a, b}
- Draw a DFA for the language accepting strings ‘ab’ in the middle (sub string) over input alphabets ∑ = {a, b}

Lecture References

- Portland State University Lectures (Link)
- Power set Construction Wikipedia (Link)
- Maynooth University Lectures (Link)

References/Books

- 1. Compilers-Principles, techniques and tools (2nd Edition) V. Aho, Sethi and D. Ullman
- 2. Principles of Compiler Design (2nd Revised Edition 2009) A. A. Puntambekar
- 3. Basics of Compiler Design Torben Mogensen

FIRST and FOLLOW

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:	9	**Week No:**	9	**Semester:**	
Lecturer:	*Md Masum Billah, billah.masumcu@aiub.edu*				

Lecture Outline

1. Review of Subset Construction Rule (NFA to DFA conversion)
2. Overview of First and Follow
3. First and Follow set Rules
4. Examples
5. Exercises

Objective and Outcome

Objective:

- To Explain the necessity or requirement of FIRST and FOLLOW set calculation.
- To elaborate the method/algorithm of FIRST and FOLLOW calculation from a given CFG.
- To provide necessary example and exercise of FIRST and FOLLOW calculation from a given CFG

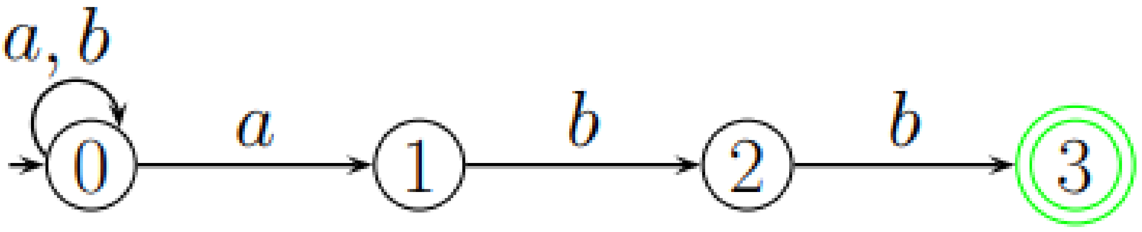
Outcome:

- After this class the students will know the necessity of FIRST and FOLLOW calculation
- After this class the students will be able to demonstrate the FIRST and FOLLOW calculation method.
- The students will also be capable of calculating FIRST and FOLLOW set from a given CFG

Review on NFA to DFA

Example

A NFA for the language, L3 = {a, b}∗{abb}.

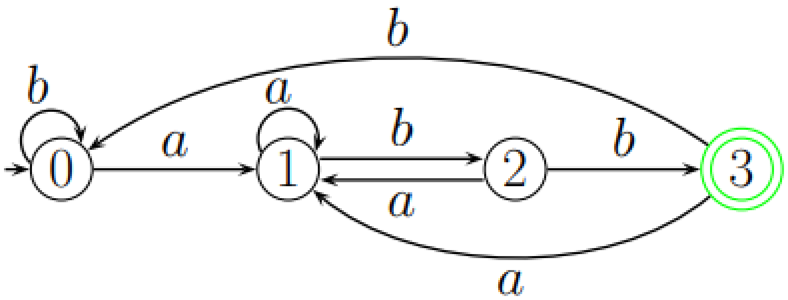


Given NFA

Review on NFA to DFA

Example

names	states	a	b
A	$\{0\}$	B	A
B	$\{0,1\}$	B	C
C	$\{0,2\}$	B	D
D	$\{0,3\}$	B	A



Converted DFA

FIRST and FOLLOW Overview

The basic problem in parsing is choosing which production rule to use at any stage during a derivation.

- Lookahead

Means attempting to analyze the possible production rules which can be applied, in order to pick the one most likely to derive the current symbol(s) on the input.

- FIRST and FOLLOW

We formalize the task of picking a production rule using two functions, FIRST and FOLLOW. we need to find FIRST and FOLLOW sets for a given grammar, so that the parser can properly apply the needed rule at the correct position.

FIRST Set Calculation

Rules

1. If X is terminal, FIRST(X) = {X}.
2. If X → ε is a production, then add ε to FIRST(X).
3. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, and ε is in all of FIRST(Y1), …, FIRST(Yk), then add ε to FIRST(X).
4. If X is a non-terminal, and X → Y1 Y2 … Yk is a production, then add a to FIRST(X) if for some i, a is in FIRST(Yi), and ε is in all of FIRST(Y1), …, FIRST(Yi-1).

Applying rules 1 and 2 is obvious. Applying rules 3 and 4 for FIRST(Y1 Y2 … Yk) can be done as follows:

Add all the non-ε symbols of FIRST(Y1) to FIRST(Y1 Y2 … Yk). If ε ∈ FIRST(Y1), add all the non-ε symbols of FIRST(Y2). If ε ∈ FIRST(Y1) and ε ∈ FIRST(Y2), add all the non-ε symbols of FIRST(Y3), and so on. Finally, add ε to FIRST(Y1 Y2 … Yk) if ε ∈ FIRST(Yi), for all 1 ≤ i ≤ k.

First Set

The algorithm to compute the firsts set of a symbol X:

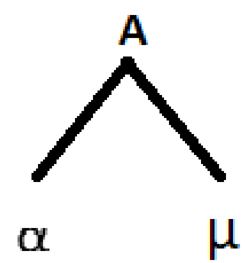
```
if(X is a terminal symbol):
  first(X) = X;
  break;
if (X -> Ɛ ∈ productions of the grammar):
  first(X).add({ Ɛ });
foreach(X -> Y1....Yn ∈ productions of the grammar):
  j = 1;
  while (j <= n):
    first(X).add({ b }), ∀ b ∈ first(Yj) ;
    if ( Ɛ ∈ first(Yj)):
       j ++;
    else:
       break;
if(j = n+1):
  first(X).add({ Ɛ });
```

First Set (Case 1)

- For a Production, if the first things is terminals that terminal (left most) would be considered as a 'First'
- If the Left most thing is a terminals then that terminals will be 'First'
- Don't worry about the rest of the things residing on the right side of the first terminals

Case 1

$A \to \alpha\mu$



$\text{FIRST}(A) = \{\alpha\}$

First Set (Case 2)

- For a Production, if the first things is epsilon (ε) then ‘FIRST’ is epsilon (ε)

First Set (Case 3)

- For a Production, if the first things is Non-Terminals, then we should continue until we found a terminals.

- Look for the next production and next until we encounter a terminals

First Set (Example 1)

Problem	Solution
E -> TE' E' -> +T E' | ϵ T -> F T' T' -> *F T' | ϵ F -> (E) | id	FIRST(E) = FIRST(T) = { (, id } FIRST(E') = { +, ϵ } FIRST(T) = FIRST(F) = { (, id } FIRST(T') = { *, ϵ } FIRST(F) = { (, id }

First Set (Example 2)

Problem

```
S -> ACB | Cbb | Ba
A -> da | BC
B -> g | Є
C -> h | Є
```

Solution

```
FIRST sets
FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C)
         = { d, g, h, Є, b, a}
FIRST(A) = { d } U FIRST(B) = { d, g , h, Є }
FIRST(B) = { g , Є }
FIRST(C) = { h , Є }
```

Follow Set

Rules

- Follow should be look for right side of anything
- Follow always starts with $
- **Follow(X)** to be the set of terminals that can appear immediately to the right of Non-Terminal X in some sentential form.
- FOLLOW (S) = { S } // where S is the starting Non-Terminal
- If A -> pBq is a production, where p, B and q are any grammar symbols, then everything in FIRST (q) except ε is in FOLLOW (B)
- If A->pB is a production, then everything in FOLLOW(A) is in FOLLOW (B)
- If A->pBq is a production and FIRST(q) contains ε, then FOLLOW (B) contains { FIRST(q) - ε} U FOLLOW (A)

Follow Set

Rules

Apply the following rules:

1. If \$ is the input end-marker, and S is the start symbol, $\$ \in FOLLOW(S)$.
2. If there is a production, $A \rightarrow \alpha B\beta$, then $(FIRST(\beta) - \varepsilon) \subseteq FOLLOW(B)$.
3. If there is a production, $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B\beta$, where $\varepsilon \in FIRST(\beta)$, then $FOLLOW(A) \subseteq FOLLOW(B)$.

Note that unlike the computation of FIRST sets for non-terminals, where the focus is on*what a non-terminal generates*, the computation of FOLLOW sets depends upon *where the non-terminal appears on the RHS of a production*

Follow Set (Case 1-a)

- Follow means something right behind of it.
- Follow means the next one
- If the next of a thing (whos Follow should be calculated) **terminal**/nonterminal then we must find the 'FIRST' of that terminal/nonterminal
- That particular 'FIRST' would be the designated 'FOLLOW' of the things (whos Follow should be calculated)

Follow Set (Case 1-b)

- Follow means something right behind of it.
- Follow means the next one
- If the next of a thing (whos Follow should be calculated) terminal/nonterminal then we must find the 'FIRST' of that terminal/nonterminal
- That particular 'FIRST' would be the designated 'FOLLOW' of the things (whos Follow should be calculated)

Follow Set (Case 2)

- We never write epsilon (ε) in ‘FOLLOW’
- If we do not have anything on right side
- That is, if we do not have an ‘FOLLOW’ then we will take the ‘FOLLOW’ (all FOLLOW) of its parent (non-terminal) (from which the production came)

Follow Set (Example 1)

Problem

```
Production Rules:
E -> TE’
E’ -> +T E’|Є
T -> F T’
T’ -> *F T’ | Є
F -> (E) | id
```

Solution

```
FIRST set
FIRST(E) = FIRST(T) = { ( , id }
FIRST(E’) = { +, Є }
FIRST(T) = FIRST(F) = { ( , id }
FIRST(T’) = { *, Є }
FIRST(F) = { ( , id }

FOLLOW Set
FOLLOW(E)  = { $ , ) }  // Note  ')' is there because of 5th rule
FOLLOW(E’) = FOLLOW(E) = {  $, ) }  // See 1st production rule
FOLLOW(T)  = { FIRST(E’) – Є } U FOLLOW(E’) U FOLLOW(E) = { + , $ , ) }
FOLLOW(T’) = FOLLOW(T) =      { + , $ , ) }
FOLLOW(F)  = { FIRST(T’) –  Є } U FOLLOW(T’) U FOLLOW(T) = { *, +, $, ) }
```

Follow Set (Example 2)

Problem

```
Production Rules:
S -> ACB|Cbb|Ba
A -> da|BC
B-> g|Є
C-> h| Є
```

Solution

```
FIRST set
FIRST(S) = FIRST(A) U FIRST(B) U FIRST(C) = { d, g, h, Є, b, a}
FIRST(A) = { d } U FIRST(B) = { d, g, Є }
FIRST(B) = { g, Є }
FIRST(C) = { h, Є }

FOLLOW Set
FOLLOW(S) = { $ }
FOLLOW(A)  = { h, g, $ }
FOLLOW(B) = { a, $, h, g }
FOLLOW(C) = { b, g, $, h }
```

First and Follow Set

Example

Grammar	First	Follow
S->ABCDE	{a, b, c}	{ $ }
A-a/epsilon	{a, epsilon}	{b, c}
B->b/epsilon	{b, epsilon}	{c}
C->c	{c}	{d, e, $}
D->d/epsilon	{d, epsilon}	{e, $ }
E->e/epsilon	{e, epsilon}	{$}

Lecture References

- Online Tool:
http://jsmachines.sourceforge.net/machines/ll1.html
- Online Tutorial
https://www.geeksforgeeks.org/why-first-and-follow-in-compiler-design/
- Maynooth University Material
http://www.cs.nuim.ie/~jpower/Courses/Previous/parsing/node48.html
- StackOverflow Explanation
https://stackoverflow.com/questions/3720901/what-is-the-precise-definition-of-a-lookahead-set

References/ Books

- 1. Compilers-Principles, techniques and tools (2nd Edition) V. Aho, Sethi and D. Ullman
- 2. Principles of Compiler Design (2nd Revised Edition 2009) A. A. Puntambekar
- 3. Basics of Compiler Design Torben Mogensen