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
ϵ-*closure*(s)	Set of NFA states reachable from NFA state s on ϵ-transitions alone.
ϵ-*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.

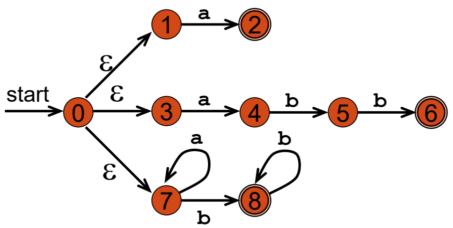
NFA to DFA Conversion

Subset Construction Algorithm

```
initially, ε-closure(s0) 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**) = $\varnothing$

Alphabet / Symbol = {a, b}

Subset Construction Algorithm

Subset Construction Algorithm

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

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

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

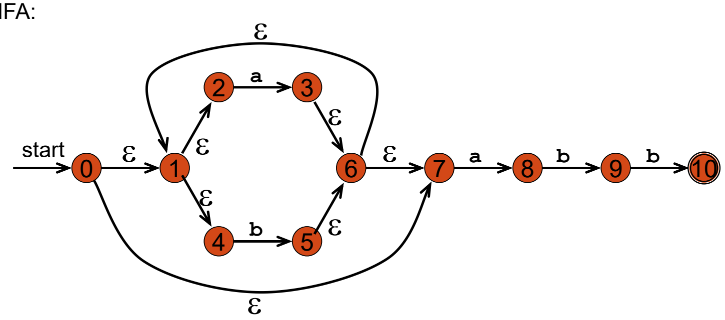
Algorithm with while Loop

```
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

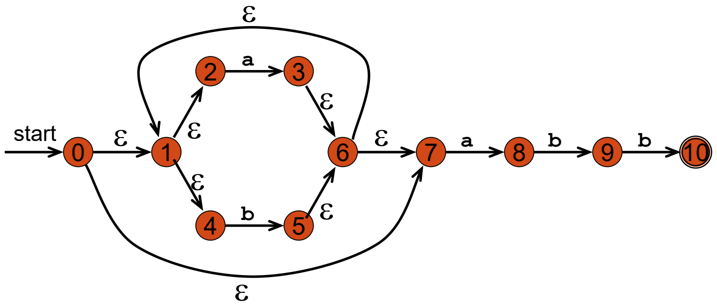
Subset Construction Method (Example-1)

NFA:



Regular Expression: (a | b)* abb

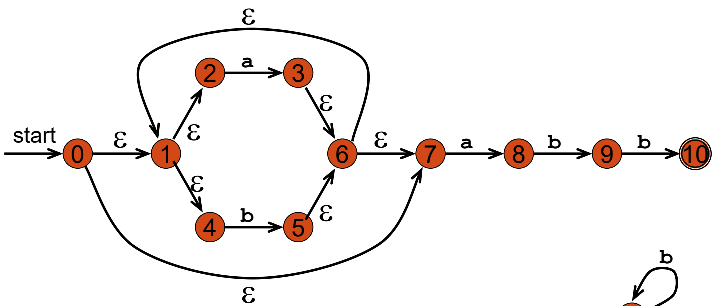
Subset Construction Method (Example-1)



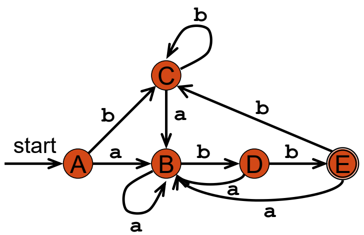
DFA State	E-closure of	E-closure outcome states
A	E-closure ({0})	0,1,2,4,7
B	E-closure ({3,8})	1,2,3,4,6,7,8
C	E-closure ({5})	1,2,4,5,6,7
D	E-closure({5,9})	1,2,4,5,6,7,9
E	E-closure({5,10})	1,2,4,5,6,7,10

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.)



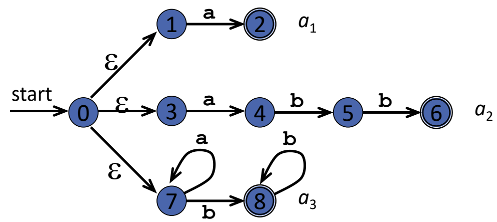
NFA State	DFA State	a	b
0,1,2,4,7	A	B	C
1,2,3,4,6,7,8	B	B	D
1,2,4,5,6,7	C	B	C
1,2,4,5,6,7,9	D	B	E
1,2,4,5,6,7,10	E	B	C



NFA to DFA Conversion

Subset Construction Method (Exercise 1)

NFA

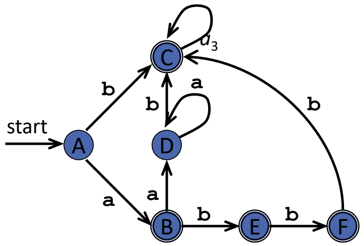


Converted DFA in the next Slide

NFA to DFA Conversion

Subset Construction Method (Exercise 1)

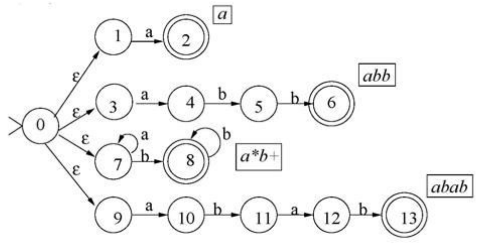
DFA



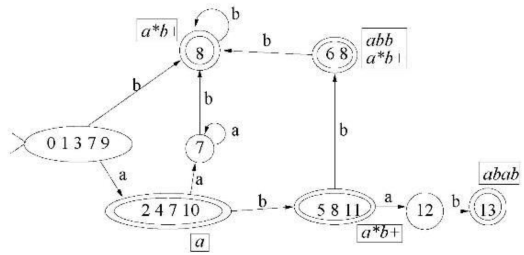
Dstates
A = {0,1,3,7}
B = {2,4,7}
C = {8}
D = {7}
E = {5,8}
F = {6,8}

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

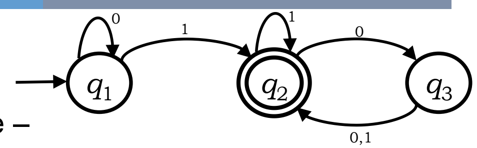


Figure: Finite Automaton M_1

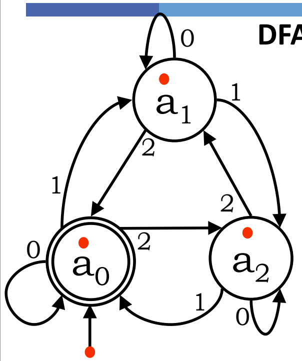
- M_1 = (Q, Σ, δ, q_0, F), where –
  - $Q = \{q_1, q_2, q_3\}$,
  - $\Sigma = \{0, 1\}$,
  - *δ is describe as –*
  - $q_0 = q_1$,
  - $F = \{q_2\}$.

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



- Alphabet $\Sigma=\{0,1,2\}$.
- 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 $M_1= (Q_1, \Sigma, \delta_1, q_1, F_1)$, 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

- Input example: 01120101
- Present State:

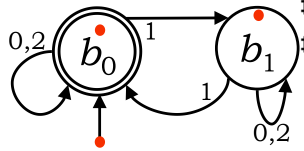
a_2

- Input symbol:

ε

Accepted

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

Lecture No:	9.1	Week No:	9	Semester:	Summer 2020-2021
Lecturer:	MAHFUJUR RAHMAN, *mahfuj@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

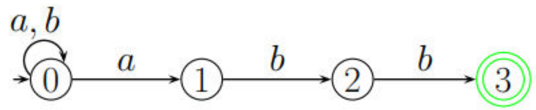
Outcome:

- 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}.

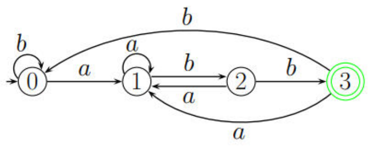


Given NFA

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:

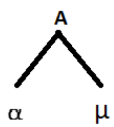
```
if(X is a terminal symbol):
  first(X) = X;
  break;
if (X -> Ɛ ∈ productions of the grammar):
  first(X).add({ Ɛ });
foreach(X -> Y1....Yn ∈ productions of the grammar):
  j = 1;
  while (j <= n):
    first(X).add({ b }), ∀ b ∈ first(Yj) ;
    if ( Ɛ ∈ first(Yj)):
      j ++;
    else:
      break;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



FIRST (A) = { α }

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

```
E  -> TE'
E' -> +T E'|є
T  -> F T'
T' -> *F T' | є
F  -> (E) | id
```

Solution

```
FIRST(E) = FIRST(T) = { ( , id }
FIRST(E') = { +, є }
FIRST(T) = FIRST(F) = { ( , id }
FIRST(T') = { *, є }
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

Parsing and Parsing Table

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**10.1**	**Week No:**	**10**	**Semester:**	**Summer 2020-2021**
Lecturer:	**MAHFUJUR RAHMAN,** ***mahfuj@aiub.edu***				

Lecture Outline

1. Parsing
2. Parsing Technique (LL1 Grammar)
3. Parsing Table Construction Technique
4. Examples
5. Exercises

Objective and Outcome

Objective:

- To provide an overview of parsing and parsing types.
- To give an overview of predictive parser
- To demonstrate the predictive parsing table construction for predictive / LL(1) parser from a given CFG

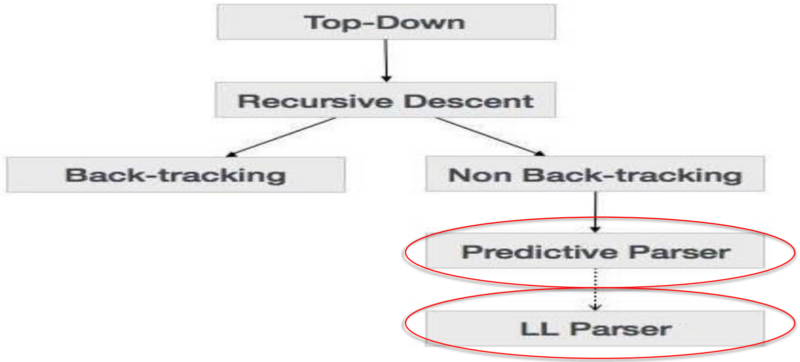
Outcome:

- After this lecture the students will be able to understand basics of predictive and LL (1) parser.
- The students will be capable of constructing a predictive parsing table from given CFG

Parsing

- The process of determining if **a string of terminals (tokens) can be generated by a grammar.**
- Time complexity:
  - For any CFG there is a parser that takes at most $O(n^3)$ time to parse a string of n terminals.
  - Linear algorithms suffice to parse essentially all languages that arise in practice.
- Two kinds of methods:
  - **Top-down:** constructs a parse tree from root to leaves
  - **Bottom-up:** constructs a parse tree from leaves to root

Types of Parsing



Parsing Table Overview

- A Parsing table collects information from **FIRST** and **FOLLOW** set.
- A Parsing table provides a direction/predictive guideline for generating a **parse tree** from a grammar.
- A Parsing table provide information to create moves made by a predictive parser on a specific input.

LL(k) LL(1) Parser Design Prerequisite

- Make the grammar **suitable for top-down parser.** By performing the **elimination of left recursion.** And by **performing left factoring.**
- Find the **FIRST** and **FOLLOW** of the **variables.**
- Create Parsing table based on the **information from FIRST and FOLLOW sets.**

Predictive (LL1) Parsing Table Construction Rule

- Collect information from FIRST and FOLLOW sets into a predictive parsing Table M[A, a]
- M[A, a] is a 2D array where
  - A nonterminal
  - A is a terminal or the symbol $, the input end-marker
- The Production A -> a is chosen if the next input symbol a is in First (a).
- If a = ε, we should again choose A-> a, if the current input symbol is in FOLLOW (A) or if the $ on the input has been reached and $ is in the FOLLOW(A)

Predictive (LL1) Parsing Table Construction Rule

- From a Grammar Find out First and Follow
- Take a production; Row should be left hand side and column should be first of right and side
- If we see epsilon in first of right hand side, place the production in follow also
- If first of right hand side terminal, directly place in table
- If the first of right hand side is epsilon, directly place in follow of left hand side

```
E  -> TE'
E' -> +T E'|ϵ
T  -> F T'
T' -> *F T' | ϵ
F  -> (E) | id
```

```
E  -> TE'
E' -> +T E'|ϵ
T  -> F T'
T' -> *F T' | ϵ
F  -> (E) | id
```

```
FIRST set
FIRST(E) = FIRST(T) = { ( , id }
FIRST(E') = { +, ϵ }
FIRST(T) = FIRST(F) = { ( , id }
FIRST(T') = { *, ϵ }
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') - ϵ } U FOLLOW(E') U FOLLOW(E) = { + , $ , ) }
FOLLOW(T') = FOLLOW(T) =      { + , $ , ) }
FOLLOW(F)  = { FIRST(T') -  ϵ } U FOLLOW(T') U FOLLOW(T) = { *, +, $, ) }
```

```
E  -> TE'
E' -> +T E'|ϵ
T  -> F T'
T' -> *F T' | ϵ
F  -> (E) | id
```

First Set	Follow Set	Variables	+	*	(	)	id	$
{ (, id }	{ $,) }	E						
{ +, ε }	{ $,) }	E’						
{ (, id }	{ +, $,) }	T						
{ *, ε }	{ +, $,) }	T’						
{ (, id }	{ *, +, $,) }	F						

Parsing Table Construction (Example)

LL(1) grammar ('' is ε):

```
E -> T E'
E' -> + T E'
E' -> ''
T -> F T'
T' -> * F T'
T' -> ''
F -> ( E )
F -> id
```

FIRST	FOLLOW	Nonterminal	+	*	(	)	id	$
{(,id}	{$,)}	E			E -> T E'		E -> T E'	
{+,''}	{$,)}	E'	E' -> + T E'			E' -> ''		E' -> ''
{(,id}	{+,$,)}	T			T -> F T'		T -> F T'	
{*,''}	{+,$,)}	T'	T' -> ''	T' -> * F T'		T' -> ''		T' -> ''
{(,id}	{*,+,$,)}	F			F -> (E)		F -> id	

Predictive parsing table for the grammar (Example 1)

$S \rightarrow +SS \mid *SS \mid a;$

FIRST(s) = {+, *, a}
FOLLOW(s) = {+, *, a, $}

Nonterminal	Input Symbol a	+	*	$
S	S → a	S → +SS	S → *SS	error

Predictive parsing table for the grammar (Example 2)

$S \rightarrow (S) S \mid \varepsilon$

FIRST(s) = {(, ε}
FOLLOW(s) = {), $}

Nonterminal	Input Symbol (	)	$
S	S → (S)S	S → ε	S → ε

Predictive parsing table for the grammar (Example 3)

$S \rightarrow S\ (\ S\ )\ |\ \varepsilon$

FIRST(s) = {(, ε}
FOLLOW(s) = {(,), $}

Nonterminal	(	)	$
S	$S \rightarrow S(S)$ $S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$	$S \rightarrow \varepsilon$

(Input Symbol)

Parsing Table Construction (Problem)

Consider the following LL(1) grammar, which has the set of terminals *T* = *f***a***;* **b***;* **ep***;* **+***;* ******;* **(***;* **)***g*. This grammar generates regular expressions over *f*a, b*g*, with **+** meaning the RegExp OR operator, and **ep** meaning the ε symbol. (Yes, this is a context free grammar for generating regular expressions!)

$$\begin{aligned} E &\rightarrow TE' \\ E' &\rightarrow +E \mid \epsilon \\ T &\rightarrow FT' \\ T' &\rightarrow T \mid \epsilon \\ F &\rightarrow PF' \\ F' &\rightarrow *F' \mid \epsilon \\ P &\rightarrow (E) \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{ep} \end{aligned}$$

Parsing Table Construction (Solution)

FIRST and FOLLOW sets

$\text{First}(E) = \{(, a, b, ep\}$	$\text{Follow}(E) = \{), \$\}$
$\text{First}(E') = \{+, \epsilon\}$	$\text{Follow}(E') = \{), \$\}$
$\text{First}(T) = \{(, a, b, ep\}$	$\text{Follow}(T) = \{+,), \$\}$
$\text{First}(T') = \{(, a, b, ep, \epsilon\}$	$\text{Follow}(T') = \{+,), \$\}$
$\text{First}(F) = \{(, a, b, ep\}$	$\text{Follow}(F) = \{(, a, b, ep, +,), \$\}$
$\text{First}(F') = \{*, \epsilon\}$	$\text{Follow}(F') = \{(, a, b, ep, +,), \$\}$
$\text{First}(P) = \{(, a, b, ep\}$	$\text{Follow}(P) = \{(, a, b, ep, +,), *, \$\}$

Parsing Table Construction (Solution)

LL (1) Parsing Table

	(	)	a	b	ep	+	*	$
E	TE'		TE'	TE'	TE'			
E'		ϵ				$+E$		ϵ
T	FT'		FT'	FT'	FT'			
T'	T	ϵ	T	T	T	ϵ		ϵ
F	PF'		PF'	PF'	PF'			
F'	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	$*F'$	ϵ
P	(E)		a	b	ep			

Lecture References

- Carnegie Mellon University Material
https://www.cs.cmu.edu/~fp/courses/15411-f09/lectures/08-predictive.pdf
- Columbia University Material
http://www1.cs.columbia.edu/~aho/cs4115/lectures/13-02-20.htm
- Online Material
https://www.ques10.com/p/8960/construct-predictive-passing-table-for-following-2/
- Online Tutorial
https://www.tutorialspoint.com/compiler_design/compiler_design_top_down_parser.htm

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

Stack Movement Predictive parser

Course Code: CSC3220 Course Title: Compiler Design

Dept. of Computer Science
Faculty of Science and Technology

Lecture No:	**11.1**	**Week No:**	**11**	**Semester:**	**Summer 2020-2021**
Lecturer:	**MAHFUJUR RAHMAN,** ***mahfuj@aiub.edu***				

Lecture Outline

1. First, Follow and Parsing Table Exercise and Practice
2. Non-Recursive predictive parsing
3. Stack Movement of Predictive parser

Objective and Outcome

Objective:

- To review predictive parsing table construction with example
- To elaborate the necessity of stack movement by a predictive parser
- To explain non-recursive predictive parsing algorithm
- Demonstrate stack movement of a predictive parser for a certain input with example

Outcome:

- The student will improve their ability of FIRST, FOLLOW and parsing table construction skills.
- After this class the students will understand non-recursive predictive parsing algorithm
- The students will be capable of demonstrating stack movement of a predictive parser for a certain given input string from given Grammar (CFG)

Predictive parsing table for the grammar (Example)

Example:

```
S-> A
A-> aB| Ad
B->bBC|f
C→g
```

Step 1:

```
A →Ad/aB
LR
A  →  aBA'
A'  →  dA'|€
S    →  A
B    →  bBC|f
C    →  g
```

Predictive parsing table for the grammar (Example)

Step 2

FIRST

(S)	**{a}**
(A)	{a}
(A')	{d, €}
(B)	(b,f)
(C)	{g}

FOLLOW

(S)	{$}
(A)	{$}
(A')	{$}
(B)	{d,g,$}
(C)	{d,g,$}

Predictive parsing table for the grammar (Example)

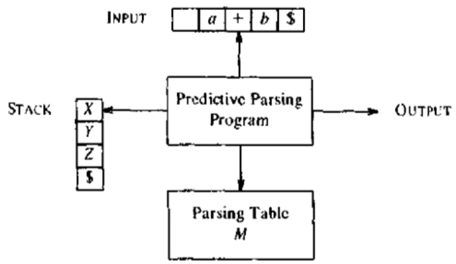
Step 3

-	a	b	d	g	f	$
S	S→A					-
A	A→aBA'					-
A'			A'→dA',є			A'→є
B		B→bBC			B→f	-
C			C→g			-

Non Recursive Predictive Parsing

Algorithm

- It is possible to build a non recursive predictive parser by maintaining a stack.
- The key problem during predictive parsing is that determining the production to be applied for a nonterminal.
- The non recursive parser looks up the production to be applied in the parsing table.



Non Recursive Predictive Parser

Algorithm

Input: A String (input) w, a parsing table M and a grammar G

Output: If w is in L(G), a leftmost derivation of w; or error

Method: Initially, the parser is in a configuration in which it has $S on the stack with S, the start symbol of G on top, and w$ in the input buffer. The program that utilizes the predictive parsing table M to produce a parse for th input

```
set ip to point to the first symbol of w$;
repeat
    let X be the top stack symbol and a the symbol pointed to by ip;
    if X is a terminal or $ then
        if X = a then
            pop X from the stack and advance ip
        else error()
    else        /* X is a nonterminal */
        if M[X, a] = X → Y1Y2 · · · Yk then begin
            pop X from the stack;
            push Yk, Yk-1, . . . , Y1 onto the stack, with Y1 on top;
            output the production X → Y1Y2 · · · Yk
        end
        else error()
until X = $    /* stack is empty */
```

Stack Movement

Non Recursive Predictive Parser Method

- With the help of FIRST, FOLLOW and associated Parse Table predictive parser makes moves
- With a certain input string the predictive parser makes the sequence of moves
- The input pointer points to the leftmost symbol of the string in the input column
- It is tracing out a leftmost derivation for the input, the productions output are those of a leftmost derivation
- The input symbols that have already been scanned, followed by the grammar symbols on the stack (from top to bottom), make up the left-sentential forms in the derivation.

Parsing Table Construction (Example 1)

LL(1) grammar ('' is ε):

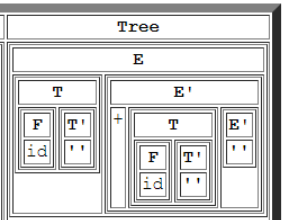
```
E -> T E'
E' -> + T E'
E' -> ''
T -> F T'
T' -> * F T'
T' -> ''
F -> ( E )
F -> id
```

FIRST	FOLLOW	Nonterminal	+	*	(	)	id	$
{(,id}	{$,)}	E			E -> T E'		E -> T E'	
{+,''}	{$,)}	E'	E' -> + T E'			E' -> ''		E' -> ''
{(,id}	{+,$,)}	T			T -> F T'		T -> F T'	
{*,''}	{+,$,)}	T'	T' -> ''	T' -> * F T'		T' -> ''		T' -> ''
{(,id}	{*,+,$,)}	F			F -> (E)		F -> id	

Stack Movement a Predictive Parser (Example 1)

Given input String: id + id

Stack	Input	Rule
`$ E`	`id + id $`	
`$ E' T`	`id + id $`	`E -> T E'`
`$ E' T' F`	`id + id $`	`T -> F T'`
`$ E' T' id`	`id + id $`	`F -> id`
`$ E' T'`	`+ id $`	
`$ E'`	`+ id $`	`T' -> ''`
`$ E' T +`	`+ id $`	`E' -> + T E'`
`$ E' T`	`id $`	
`$ E' T' F`	`id $`	`T -> F T'`
`$ E' T' id`	`id $`	`F -> id`
`$ E' T'`	`$`	
`$ E'`	`$`	`T' -> ''`
`$`	`$`	`E' -> ''`



Parsing Table Construction (Example 2)

Consider the following LL(1) grammar, which has the set of terminals *T* = *f***a**; **b**; **ep**; **+**; *****; **(**; **)***g*. This grammar generates regular expressions over *f*a, b*g*, with **+** meaning the RegExp OR operator, and **ep** meaning the ε symbol. (Yes, this is a context free grammar for generating regular expressions!)

$$
\begin{array}{rcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +E \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & T \mid \epsilon \\
F & \rightarrow & PF' \\
F' & \rightarrow & *F' \mid \epsilon \\
P & \rightarrow & (E) \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{ep}
\end{array}
$$

Parsing Table Construction (Example 2)

FIRST and FOLLOW sets

$$
\begin{array}{ll}
\text{First}(E) = \{(, a, b, ep\} & \text{Follow}(E) = \{), \$\} \\
\text{First}(E') = \{+, \epsilon\} & \text{Follow}(E') = \{), \$\} \\
\text{First}(T) = \{(, a, b, ep\} & \text{Follow}(T) = \{+,), \$\} \\
\text{First}(T') = \{(, a, b, ep, \epsilon\} & \text{Follow}(T') = \{+,), \$\} \\
\text{First}(F) = \{(, a, b, ep\} & \text{Follow}(F) = \{(, a, b, ep, +,), \$\} \\
\text{First}(F') = \{*, \epsilon\} & \text{Follow}(F') = \{(, a, b, ep, +,), \$\} \\
\text{First}(P) = \{(, a, b, ep\} & \text{Follow}(P) = \{(, a, b, ep, +,), *, \$\}
\end{array}
$$

Parsing Table Construction (Example 2)

LL (1) Parsing Table

	(	)	a	b	ep	+	*	\$
E	TE'		TE'	TE'	TE'			
E'		ϵ				$+E$		ϵ
T	FT'		FT'	FT'	FT'			
T'	T	ϵ	T	T	T	ϵ		ϵ
F	PF'		PF'	PF'	PF'			
F'	ϵ	ϵ	ϵ	ϵ	ϵ	ϵ	$*F'$	ϵ
P	(E)		a	b	ep			

Parsing Table Construction (Example 2)

operation of an LL(1) parser on the input string ab*.

Stack	Input	Action
$E\$$	$ab*\$$	TE'
$TE'\$$	$ab*\$$	FT'
$FT'E'\$$	$ab*\$$	PF'
$PF'T'E'\$$	$ab*\$$	a
$aF'T'E'\$$	$ab*\$$	$terminal$
$F'T'E'\$$	$b*\$$	ϵ
$T'E'\$$	$b*\$$	T
$TE'\$$	$b*\$$	FT'
$FT'E'\$$	$b*\$$	PF'
$PF'T'E'\$$	$b*\$$	b
$bF'T'E'\$$	$b*\$$	$terminal$
$F'T'E'\$$	$*\$$	$*F'$
$*F'T'E'\$$	$*\$$	$terminal$
$F'T'E'\$$	$\$$	ϵ
$T'E'\$$	$\$$	ϵ
$E'\$$	$\$$	ϵ
$\$$	$\$$	$ACCEPT$

Lecture References

- Compilers-Principles, techniques and tools (2nd Edition) V. Aho, Sethi and D. Ullman

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