Lecture 2

Compiler: is a software translates programs in a source language into target language.

Interpreter: does not produce a target program, It reads an executable program and produces the results.

Hybrid Compilers: Source program is compiled into an intermediate program, which is later interpreted by an interpreter.

Lexical analyzer (scanner): Reads source program as a stream of characters (Char by Char from left-to-right) and classify it into tokens.

Token Types: Key words, Identifiers, Relop, Num, Op, White spaces.

Regular expression: expression that matches sets of strings.

Source Code -> Scanner -> Tokens

Tokens -> Parser -> Syntax Tree

Syntax Tree -> Semantic Analyzer -> Annotated Tree

Annotated Tree -> Source Code Optimizer -> Intermediate Code

Intermediate Code -> Code Generator -> Target Code

Target Code -> Target Code Optimizer -> Target Code

Lecture 3, 4

Finite Automata consists of:

1- S -> Set of states   S = {q0, q1, …}

2- $\sum$ -> Set of symbols $\sum$ =

State	Input 1	Input 2

3- δ -> Transition function

4- s_0 -> Start state s_0 = {q0}

5- F -> Accepting state   F = {q5, q6, …}

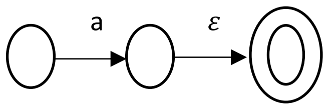
Scanner phases:

1- Pattern specification using regular expression.
2- Pattern recognition using finite automata.

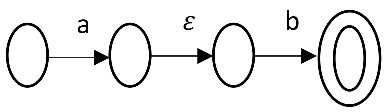
1- Thompson's construction (RE -> NFA)

Convert Regular expression to non-deterministic finite automata

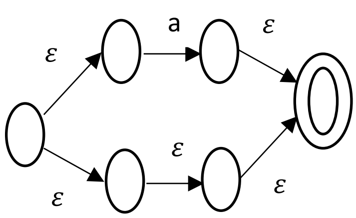
1- a



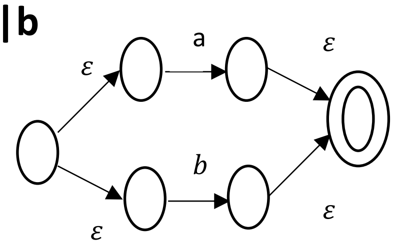
4- ab



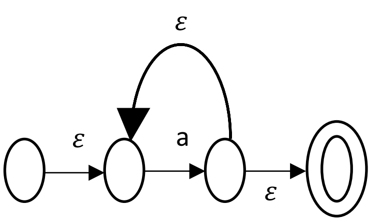
2- a?



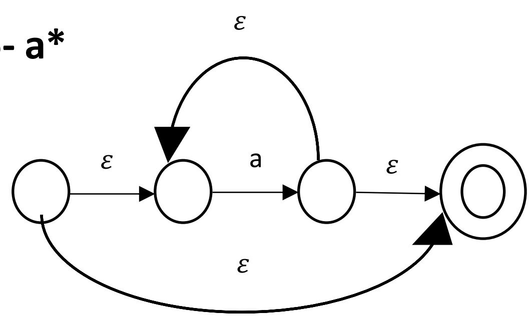
5- a|b



3- a+



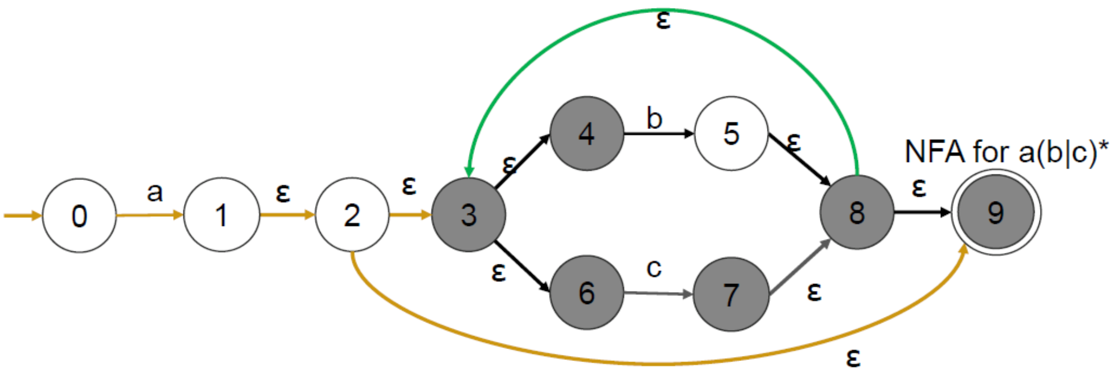
6- a*



Lecture 5

2- Subset construction (NFA -> DFA)

Convert non-deterministic finite automata to deterministic finite automata.



NFA State	a	b	c
d0 = 0	d1 = {1,2,3,4,6,9}	none	none
d1 = {1,2,3,4,6,9}	none	d2 = {3,4,5,6,8,9}	d3 = {3,4,6,7,8,9}
d2 = {3,4,5,6,8,9}	none	d2 = {3,4,5,6,8,9}	d3 = {3,4,6,7,8,9}
d3 = {3,4,6,7,8,9}	none	d2 = {3,4,5,6,8,9}	d3 = {3,4,6,7,8,9}

3- Hopcroft's algorithm (Mini DFA)

Minimizing deterministic finite automata

a- Remove unreachable state -> cannot reach it using any input.

b- Merge equivalent state.

Partition	Set	Input	Action
{final group} {non final group}	{non final group}	Input test	Split {}
{final group} {non final group}	{non final group}	All inputs	None

c- Remove dead state -> not final state that take input and it not have output.

4- DFA -> Code (code, FLex)

1- a. using doubly nested case analysis

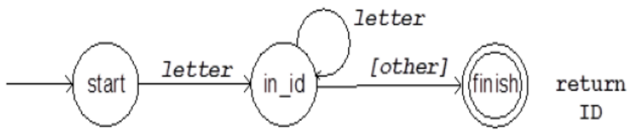
```
state = start;
getchar(input);

while(state != finish || state != error)
{
    switch (state)
    {
        case start:
            if( isalpha(input) )
            {
                advance(input);
                state = in_id;
            }
            else state = error;
            break;

        case in_id:
            if( isalpha(input) ) advance(input);
            else state =
            break;

        default: break;
    }
}

if( state == finish ) return ID;
else return error;
```



1- b. using transition table

```
i = 0;
state = 0;
while ( input[i] )
{
    state = DFA[ state , input[i++] ];
}
```

2- Fast lex (Flex)

RegExp (lex code) -> flex -> Lex.yy.c -> c compiler -> Lex.yy.exe

lex code:

```
1-  global variables
%{

    int index = 0;
%}

2- define variables
id [a-zA-Z0-9_]+
digit [0-9]

3- run commands
%%

{id}     { print(" <%s , id> " , yylex); }
{digit}  { print(" <%s , digit> " , yylex); }
"=="     { print(" <== , equal> "); }

%%


4- functions
int yywrap()
{
    return 1;
}

int main()
{
    yylex();
    return 0;
}
```

Lecture 6,7

- Any language that requires unbounded counting cannot be represented by regular language. (token -> parser -> syntax tree)

- Parser (syntax analyzer) is responsible for syntax errors.

- Grammar called:

BNF (Backus-Naur Form) or EBNF (Extended Backus-Naur Form).

- Context Free Grammars (CFG) using in parser define as:

G = (N, T, S, P)

N-> non terminal (symbol have rule)

T-> terminal (symbol have no rule)

S-> start symbol

P-> set of Production rules $\{ \alpha \rightarrow \beta \mid \alpha \in N \wedge \beta \in (N \cup T)^* \}$

- Derivation have two types (Left Most Derivation, Right Most Derivation)

LMD -> expand the leftmost nonterminal in the production, it's using with (Top-Down parser)

RMD -> expand the rightmost nonterminal in the production, it's using with (Bottom-Up parser)

- Trees have two types (parse tree (All details), syntax tree (less details))

ex:

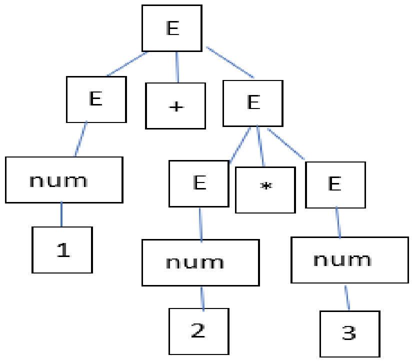
E-> E+E | E*E | num   check exp-> 1+2*3

Solve:

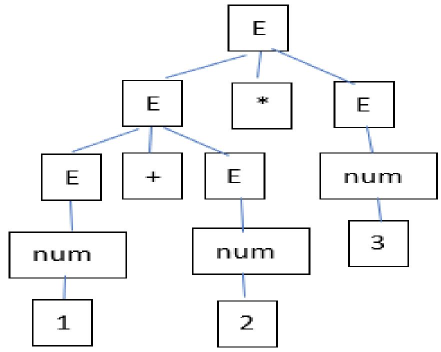
LMD		RMD	
E = E + E	E = E * E	E = E + E	E = E * E
E = num + E	E = E + E * E	E = E + E * E	E = E * num
E = 1 + E	E = num + E * E	E = E + E * num	E = E * 3
E = 1 + E * E	E = 1 + E * E	E = E + E * 3	E = E + E * 3
E = 1 + num * E	E = 1 + num * E	E = E + num * 3	E = E + num * 3
E = 1 + 2 * E	E = 1 + 2 * E	E = E + 2 * 3	E = E + 2 * 3
E = 1 + 2 * num	E = 1 + 2 * num	E = num + 2 * 3	E = num + 2 * 3
E = 1 + 2 * 3	E = 1 + 2 * 3	E = 1 + 2 * 3	E = 1 + 2 * 3

- NOTE: Trees different by starting rule not by LMD and RMD

E = E + E



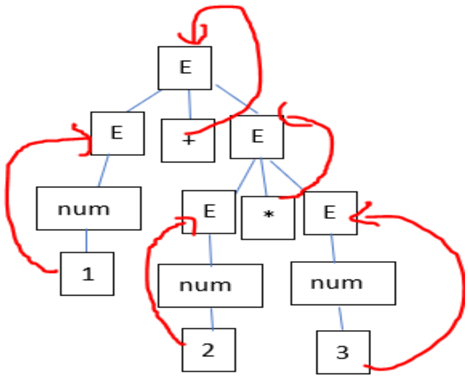
E = E * E



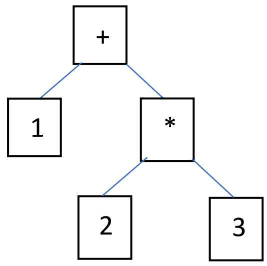
Convert parse tree to syntax tree

- **Every node that have single child (junk nodes), its child rewrite instead.**

Ex:



Parse Tree



Abstract Syntax Tree

Resolve ambiguous rules:

- **Every operation has same precedence must have non terminal.**
- **Concatenate each rule with next.**
- **Low precedence come first.**
- **Remove left recursion or right recursion according to operation associativity (+, -, /, * from left to right) (power from right to left).**

Ex: (Correct Grammar)

E -> E + T | E – T | T

T -> T * P | T / P | P

P -> N ^ P | N

N -> num

Note: to show if this grammar is ambiguous, it must draw two different parse trees for the same string that user input using the same grammar.

Lecture 8

Types of parse

- **Top – down parser:**
    - **Predictive parsing (LL1 -> Left lookahead 1 token)**

  LL1 problems

  left factoring: S-> SAB|z (temp solution: ignore)

  left recursion: S-> +A|+B|-C (Not allowed)

- **Bottom – up parser:**

First calculations: (calculate first from down to top)

- First of any terminal is itself.
- All first terminal from each exp in each |.
- If there is non-terminal take its first.
- If any first of non-terminal have epsilon then take first of next one.
- If reached to last non-terminal and its first has epsilon then epsilon will be in result.
- Duplication not allowed.

Follow calculations: (calculation of terminal and non-terminal is the same)

- Follow of start symbol must start with $.
- Find follow for each non-terminal by finding it in all expressions and take first of next symbol after it.
- If first of next symbol has ε then take first of next symbol.
- If there is no next take follow of left non-terminal.
- Duplication and epsilon are not allowed.

Ex:

S-> ABC|z|a|b
A-> a|ε
B-> b|ε
C-> cBx|ε

Solu:

Follow S = {$}
Follow A = {first B} = {b, first C} = {b, c, follow S} = {b, c, $}
Follow B = {first C, x} = {c, follow S, x} = {c, $, x}
Follow C = {follow S} = {$}

Parsing table

- For each first of non-terminal intersection between this non-terminal and every terminal in first, write rule in table related to terminal until |
- If terminal have ε write ε rule at intersect of follow terminals and non-terminal
  - Ex:

Rules: E -> TX X -> +E|ε T -> (E)|int Y Y -> *T|ε

- First E = {(, int} - Follow E = {$,)}
- First X = {+, ε} - Follow X = {$,)}
- First T = {(, int} - Follow T = {+, $,)}
- First Y = {*, ε} - Follow Y = {+, $,)}

Non terminal	Terminal					
	Int	*	+	(	)	$
E	E -> TX			E -> TX		
X			X -> +E		X -> ε	X -> ε
T	T -> int Y			T -> (E)		
Y		Y -> *T	Y -> ε		Y -> ε	Y -> ε

Build Stack from parsing table

- Put $ in stack and start symbol in stack column.
- Put input in input column.
- Take most right symbol in stack column and most left symbol in input column and get the rule in intersection between them from parsing table and make replace in reverse order of symbol in stack column with this rule (write this replace in action column).
- If most right symbol in stack column and most left symbol in input column are the same match them with each other and take pop action.
- Do that again until stack empty (Accept) or get error (Reject).
- If there is a symbol still in stack column this mean that user have not been write this symbol (like ;), and If there is a symbol still in input column this mean that user have been write this symbol without needing it.
- Note:
  - grammar in stack and input in queue.
  - Multiple entry in (LL1) parsing table not allowed (more one rule in the same cell in table)

Ex: match this input int*int$

Stack	Input	Action
$E	int*int$	Replace E -> TX
$XT	int*int$	Replace T -> int Y
$XY int	int*int$	Match (pop int)
$XY	*int$	Replace Y -> *T
$XT*	*int$	Match (pop *)
$XT	int$	Replace T -> int Y
$XY int	int$	Match (pop int)
$XY	$	Replace Y -> ε
$X	$	Replace X -> ε
$	$	Accept