**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.**

**Lecture 3, 4**

**Finite Automata consists of:**

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

**State Input 1 Input 2**

1. **-> Set of symbols = {a, b, c, …}**
2. **-> Transition function**
3. **s0 -> Start state s0 = {q0}**
4. **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**

a

b

a

**1- a 4- ab**

**2- a? 5- a|b**

a

a

a

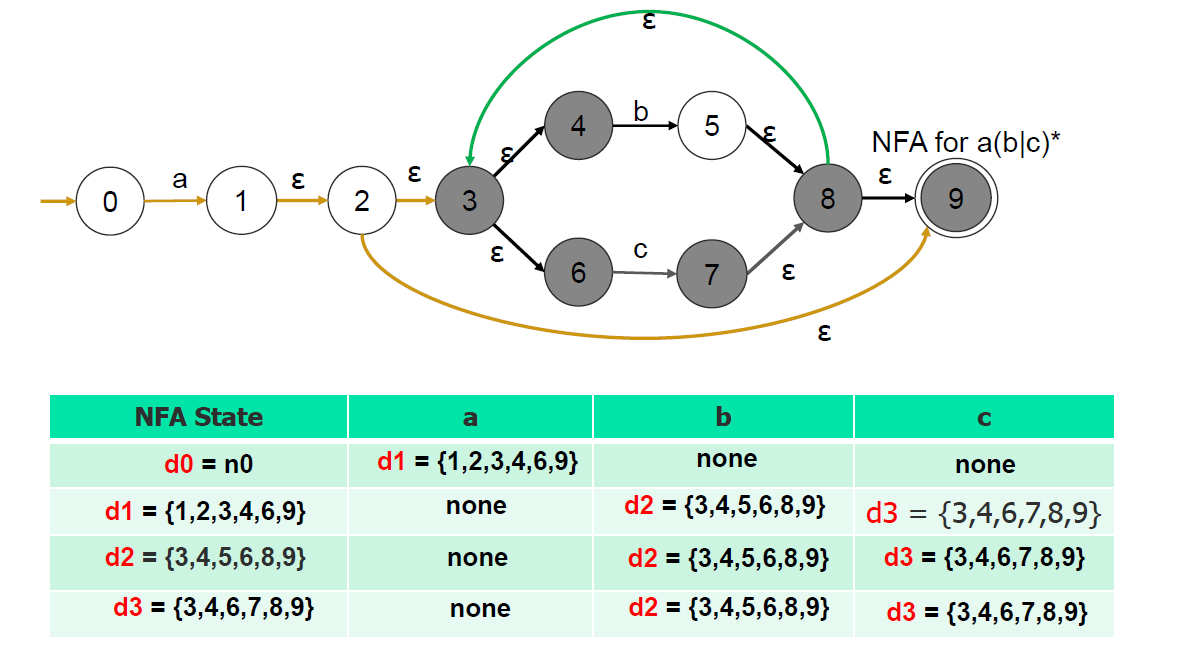
**3- a+ 6- a\***

a

**Lecture 5**

**2- Subset construction (NFA -> DFA)**

**Convert non-deterministic finite automata to deterministic finite automata.**



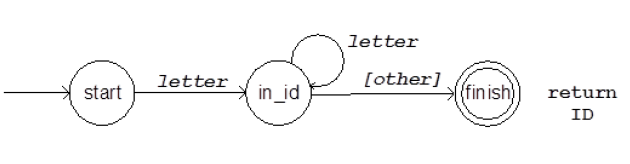
1. **Hopcroft’s algorithm (Mini DFA)**

**Minimizing deterministic finite automata**

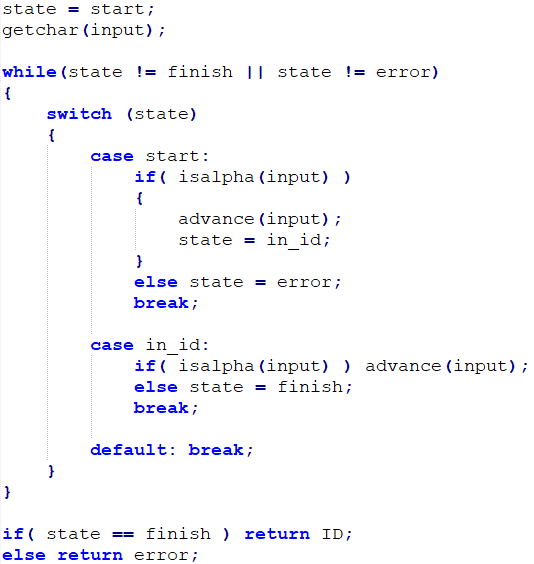
1. **Remove unreachable state -> cannot reach it using any input.**
2. **Merge equivalent state.**

|  |  |  |  |
| --- | --- | --- | --- |
| **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** |

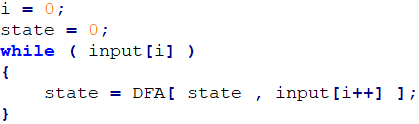
1. **Remove dead state -> not final state that take input and it not have output.**

**4- DFA -> Code (code, FLex)**

1. **a. using doubly nested case analysis**

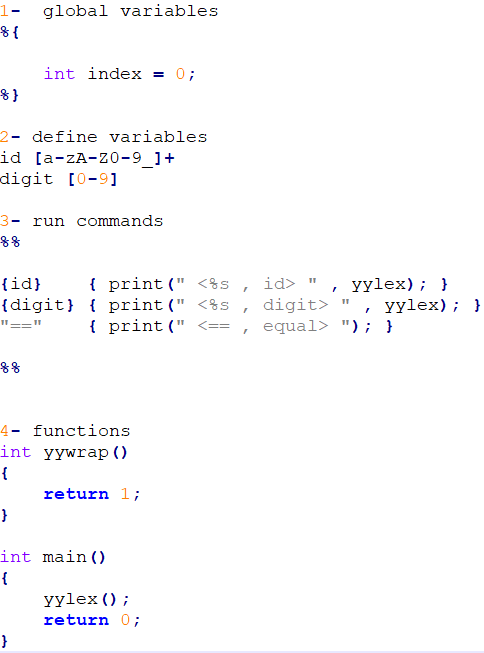
****

1. **b. using transition table**

****

**2- Fast lex (Flex)**

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

**lex code:**

**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 { α → 𝛽 | α ∈ N ∧ 𝛽 ∈ (N ∪ 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**

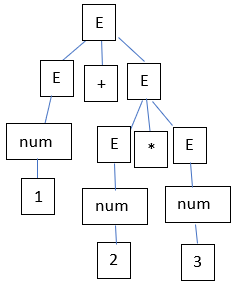
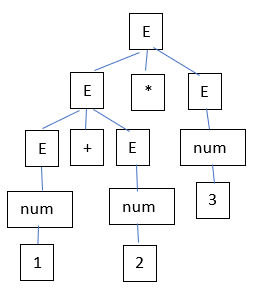
**Solve:**

**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** |

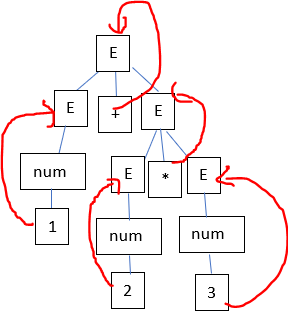
**- NOTE: Trees different by starting rule not by LMD and RMD**

**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:**

+

\*

1

2

3

Abstract Syntax Tree

Parse 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 recursion: S-> SAB|z (temp solution: ignore)**

**left factoring: 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**

* **Stack column must start with $ and start symbol and input must end with $.**
* **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** |

**Lecture 9**

**Bottom – up parser (called shift - reduce parser) advantages:**

* **More powerful (left recursion is not a problem).**
* **Using in practical (like LR1 parser).**
* **Too complex for hand coding.**
* **Begin at leaves until reach root (reduction process).**

**Bottom – up parser types:**

* **LR (0) -> L is left scanning, R is right derivation & 0 is without lookahead.**
* **SLR (1) -> simple of LR (1) and it’s an improvement of LR (0).**
* **LR (1) -> most general bottom – up algorithm.**
* **LALR (1) -> LA is lookahead, more powerful than SLR (1) and less complex than LR (1).**

**Notes:**

* **Augmented rule: any non-terminal lead to first non-terminal.**
* **Shift rule: dot doesn’t come at end S -> (L.)**
* **Reduce rule: dot come at end S’ -> S.**

**We will study LR0 parser**

**LR0 problems**

* **Shift - Reduce conflict: shift rule with reduce rule in the same box.**
* **Reduce - Reduce conflict: more than one reduce rule in the same box.**

**Build Finite automata**

**Ex:**

**Production rules: (given)**

**S -> (L) S -> x L -> S L -> L, S**

**Augmented rule: (solved)**

**S’ -> S**

* **Start box contain augmented rule and all rules belongs to non-terminal symbol in augmented rule and must start with dot.**

S’ -> .S  
S -> .(L)  
S -> .x

* **For each symbol come after dot in each rule  
  take this symbol and write effect on dot at this rule  
  and write all rules for all non-terminal came after dot.**

**Note:  
all blue boxes have shift rules.  
all green boxes have reduce rules.**

S

S’ -> S.

S’ -> .S  
S -> .(L)  
S -> .x

x

S -> x.

x

(

S -> (.L)  
  
L -> .S  
L -> .L, S  
  
S -> .(L)  
S -> .x

L -> S.

S

(

L -> L, S.

(

L -> L, .S  
  
S -> .(L)  
S -> .x

,

)

S -> (L).

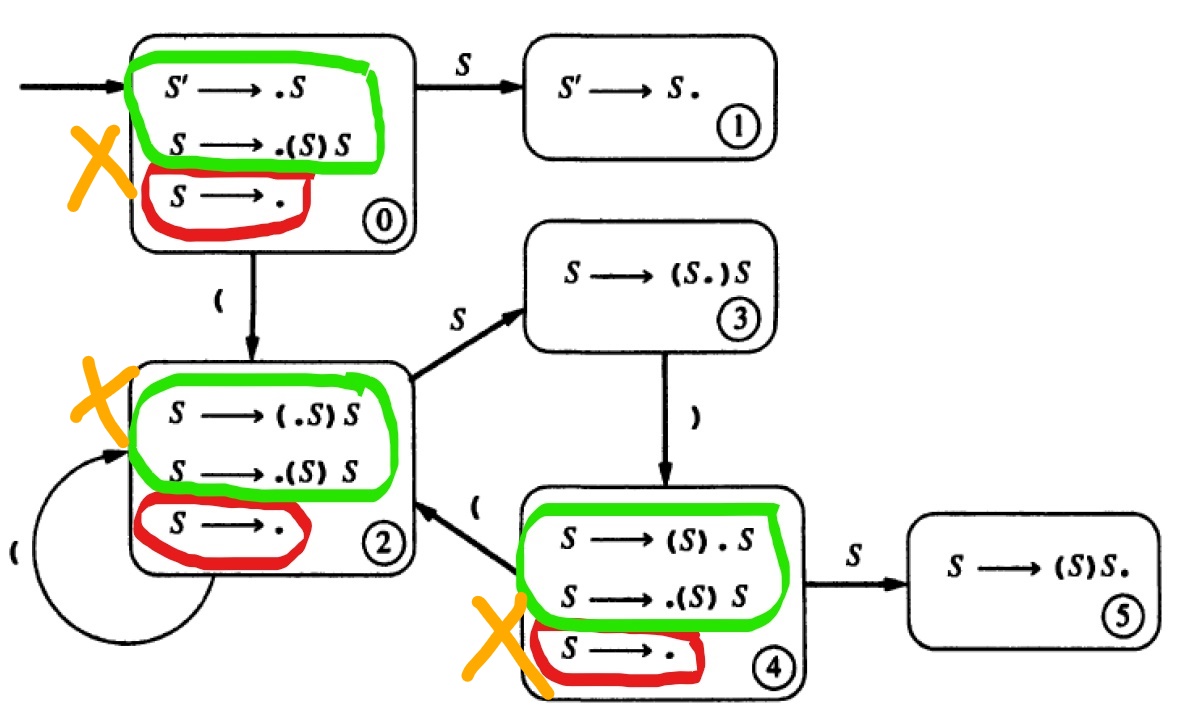
x

S

S -> (L.)  
L -> L., S

L

* **LR (0) doesn’t accept:**
  + **Reduce rule and shift rule at the same box.**
  + **More than one reduce rule at the same box.**

****

**Build Parsing table**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **State** | **Action** | **Rule** | **Input (All terminal)** | | | | **Go to (All non-terminal)** | |
| **(** | **)** | **x** | **,** | **S** | **L** |
| **0** | **Shift** |  | **3** |  | **2** |  | **1** |  |
| **1** | **Reduce** | **S’ -> S.** |  |  |  |  |  |  |
| **2** | **Reduce** | **S -> x.** |  |  |  |  |  |  |
| **3** | **Shift** |  | **3** |  | **2** |  | **4** | **5** |
| **4** | **Reduce** | **L -> S.** |  |  |  |  |  |  |
| **5** | **Shift** |  |  | **6** |  | **7** |  |  |
| **6** | **Reduce** | **S -> (L).** |  |  |  |  |  |  |
| **7** | **Shift** |  | **3** |  | **2** |  | **8** |  |
| **8** | **Reduce** | **L -> L, S.** |  |  |  |  |  |  |

**Build Stack from automata or parsing table**

* **Stack column must start with $ and 0 (start state number) and input must end with $.**
* **Match most left token from input with right most from stack and get state number.**
* **In action column write state type that at top of stack.**
* **In stack column write matched token then state number.**
* **If top of stack has reduce state then replace it with its rule and after substitution match two top symbol in stack with each other and get state number then write it at stack top.**
* **Don’t write any Dots at action column.**
* **If reduce rule get augmented rule write accept.**

|  |  |  |
| --- | --- | --- |
| **Stack** | **Input** | **Action** |
| **$0** | **(x,(x))$** | **Shift 3** |
| **$0(3** | **x,(x))$** | **Shift 2** |
| **$0(3x2** | **,(x))$** | **Reduce S -> x** |
| **$0(3S4** | **,(x))$** | **Reduce L -> S** |
| **$0(3L5** | **,(x))$** | **Shift 7** |
| **$0(3L5,7** | **(x))$** | **Shift 3** |
| **$0(3L5,7(3** | **x))$** | **Shift 2** |
| **$0(3L5,7(3x2** | **))$** | **Reduce S -> x** |
| **$0(3L5,7(3S4** | **))$** | **Reduce L -> S** |
| **$0(3L5,7(3L5** | **))$** | **Shift 6** |
| **$0(3L5,7(3L5)6** | **)$** | **Reduce S -> (L)** |
| **$0(3L5,7S8** | **)$** | **Reduce L -> L, S** |
| **$0(3L5** | **)$** | **Shift 6** |
| **$0(3L5)6** | **$** | **Reduce S -> (L)** |
| **$0S1** | **$** | **Accept** |

**Lecture 10**

**Errors types:**

* **Syntactic errors: violet grammar rules and caught by parser.**
* **Runtime errors: division by zero.**
* **Static Semantic errors: identifiers are not declared caught by semantic analyzer**
* **Semantic errors:**