IT250 Automata and Compiler Design (3-0-2)

Course Information

- Instructor : Dr. Prakash Raghavendra
  - Email: srp1970@gmail.com
  - MTech/IITM, PhD (IISc/CSA)
- Classes:
  - Online (2 hours for fortnight)
  - ~36 contact hours of teaching (Jan-April)
- Evaluation:
  - Three projects (10 * 3 = 30)
  - Mid Sem (30)
  - End Sem (40)
- Text Books:
  - Compilers: Principles, Techniques and Tools, by Aho, Sethi and Ullman, Pearson Education
  - Compiler Design in C by Allen Holub, Prentice Hall

Course Plan

Week	Plan	Remarks
Week 1, 2 and 3 (Jan 1-Jan 22)	• **Introduction to Compilers** • **Lexical Analysis:** • Regular Expressions -DFA, NFA • LEX specification	**Assignment #1**

Week 4,5 &6 (Jan 25-Feb 12)	• **Syntax Analysis**
	• **Parsing Techniques:**
	    • Top down parsing
	    • Bottom up parsing
		MidSem (Feb 15-20)

Week 7, 8 and 9 (Feb 15-Mar 5)	• **Syntax Directed Translations** • **Type Checking**	**Assignment #2**
Week 10 & 11 (Mar 8 – Mar 19)	• **Intermediate code generation** • **Runtime environment**	
Week 12, 13, 14 (Mar 22 – Apr 9)	• **Code Generation** • **Code optimization**	Apr 13 – Classes end **Assignment #3**
Apr 15-Apr 30	**End Sem Exams**	Apr 24-25 (Evaluation and checking of answer sheets)

Course Outline

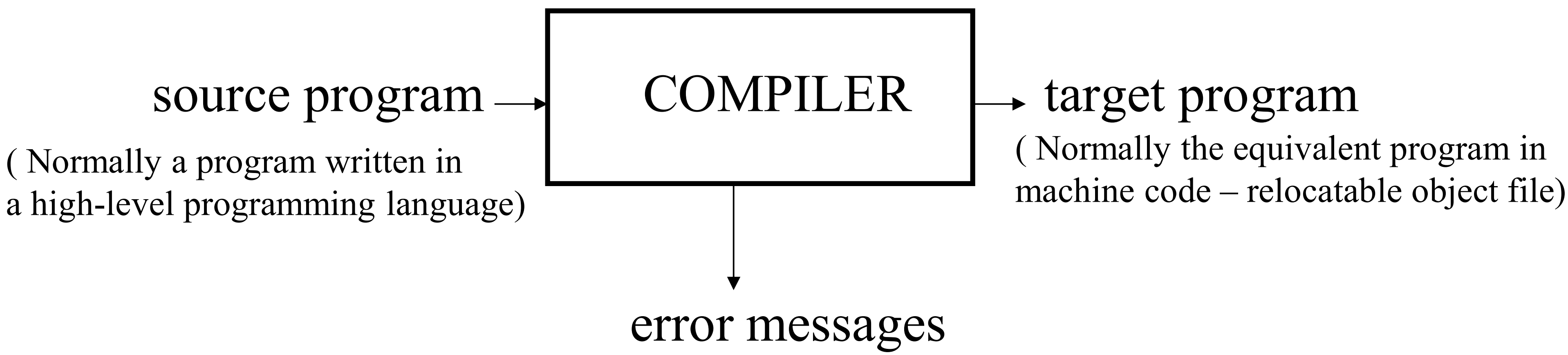
- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation
- Code Optimization

Assignment #1, #2

Assignment #3

COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



Other Applications

- In addition to the development of a compiler, the techniques used in compiler design can be applicable to many problems in computer science.
  - Techniques used in a lexical analyzer can be used in text editors, information retrieval system, and pattern recognition programs.
  - Techniques used in a parser can be used in a query processing system such as SQL.
  - Many software having a complex front-end may need techniques used in compiler design.
    - A symbolic equation solver which takes an equation as input. That program should parse the given input equation.
  - Most of the techniques used in compiler design can be used in Natural Language Processing (NLP) systems.

Major Parts of Compilers

- There are two major parts of a compiler: **Analysis** and **Synthesis**

- In analysis phase, an intermediate representation is created from the given source program.
  - Lexical Analyzer, Syntax Analyzer and Semantic Analyzer are the parts of this phase.
- In synthesis phase, the equivalent target program is created from this intermediate representation.
  - Intermediate Code Generator, Code Generator, and Code Optimizer are the parts of this phase.

Phases of A Compiler

Source Program → Lexical Analyzer → Syntax Analyzer → Semantic Analyzer → Intermediate Code Generator → Code Optimizer → Code Generator → *Target Program*

- Each phase transforms the source program from one representation into another representation.

- They communicate with error handlers.

- They communicate with the symbol table.

- A pass of compiler is when compiler reads the input once

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character and returns the *tokens* of the source program.
- A *token* describes a pattern of characters having same meaning in the source program. (such as identifiers, operators, keywords, numbers, delimiters and so on)

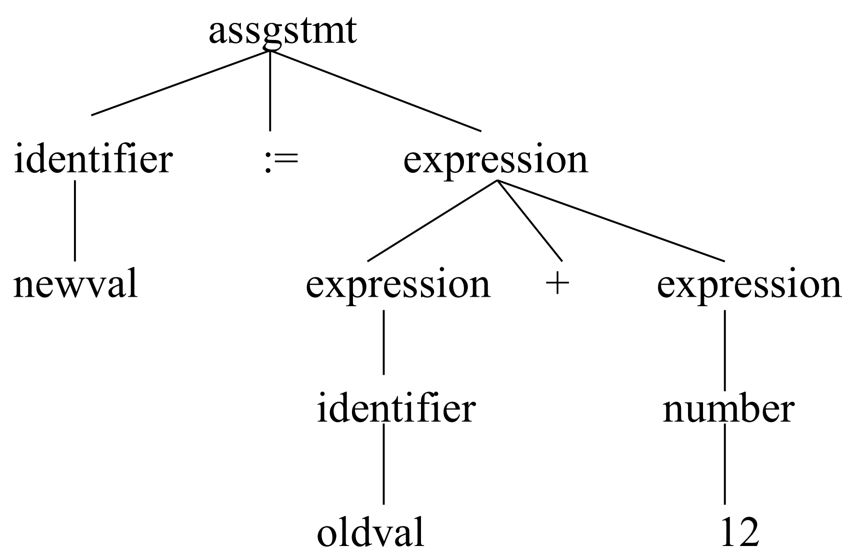
  Ex: newval := oldval + 12 => tokens:

newval	identifier
:=	assignment operator
oldval	identifier
+	add operator
12	a number

- Puts information about identifiers into the symbol table.
- Regular expressions are used to describe tokens (lexical constructs).
- A (Deterministic) Finite State Automaton can be used in the implementation of a lexical analyzer.

Syntax Analyzer

- A **Syntax Analyzer** creates the syntactic structure (generally a parse tree) of the given program.
- A syntax analyzer is also called as a **parser**.
- A **parse tree** describes a syntactic structure.



- In a parse tree, all terminals are at leaves.
- All inner nodes are non-terminals in a context free grammar.

Syntax Analyzer (CFG)

- The syntax of a language is specified by a **context free grammar** (CFG).
- The rules in a CFG are mostly recursive.
- A syntax analyzer checks whether a given program satisfies the rules implied by a CFG or not.
  - If it satisfies, the syntax analyzer creates a parse tree for the given program.

- Ex: We use BNF (Backus Naur Form) to specify a CFG

```
assgstmt    ->  identifier  := expression
expression  ->  identifier
expression  ->  number
expression  ->  expression  +  expression
```

Syntax Analyzer versus Lexical Analyzer

- Which constructs of a program should be recognized by the lexical analyzer, and which ones by the syntax analyzer?
  - Both of them do similar things; But the lexical analyzer deals with simple non-recursive constructs of the language.
  - The syntax analyzer deals with recursive constructs of the language.
  - The lexical analyzer simplifies the job of the syntax analyzer.
  - The lexical analyzer recognizes the smallest **meaningful units (tokens)** in a source program.
  - The syntax analyzer works on the smallest meaningful units (tokens) in a source program to recognize **meaningful structures** in our programming language.

Parsing Techniques

- Depending on how the parse tree is created, there are different parsing techniques.
- These parsing techniques are categorized into two groups:
  - ***Top-Down Parsing,***
  - ***Bottom-Up Parsing***
- **Top-Down Parsing:**
  - Construction of the parse tree starts at the root, and proceeds towards the leaves.
  - Efficient top-down parsers can be easily constructed by hand.
  - Recursive Predictive Parsing, Non-Recursive Predictive Parsing (LL Parsing).
- **Bottom-Up Parsing:**
  - Construction of the parse tree starts at the leaves, and proceeds towards the root.
  - Normally efficient bottom-up parsers are created with the help of some software tools.
  - Bottom-up parsing is also known as shift-reduce parsing.
  - Operator-Precedence Parsing – simple, restrictive, easy to implement
  - LR Parsing – much general form of shift-reduce parsing, LR, SLR, LALR

Semantic Analyzer

- A semantic analyzer checks the source program for semantic errors and collects the type information for the code generation.
- Type-checking is an important part of semantic analyzer.
- Normally semantic information cannot be represented by a context-free language used in syntax analyzers.
- Context-free grammars used in the syntax analysis are integrated with attributes (semantic rules)
  - the result is a syntax-directed translation,
  - Attribute grammars
- Ex:

  newval := oldval + 12

  - The type of the identifier *newval* must match with type of the expression *(oldval+12)*

Intermediate Code Generation

- A compiler may produce an explicit intermediate codes representing the source program.
- These intermediate codes are generally machine (architecture) independent. But the level of intermediate codes is close to the level of machine codes.
- Ex:

```
newval  :=  oldval * fact + 1
            ↓
id1  :=  id2 * id3 + 1
            ↓
MULT    id2,id3,temp1
ADD     temp1,#1,temp2
MOV     temp2,id1
```

Intermediates Codes (Quadraples)

Code Optimizer (for Intermediate Code Generator)

- The code optimizer optimizes the code produced by the intermediate code generator in the terms of time and space.

- Ex:

```
MULT    id2,id3,temp1
ADD     temp1,#1,id1
```

Code Generator

- Produces the target language in a specific architecture.
- The target program is normally is a relocatable object file containing the machine codes.

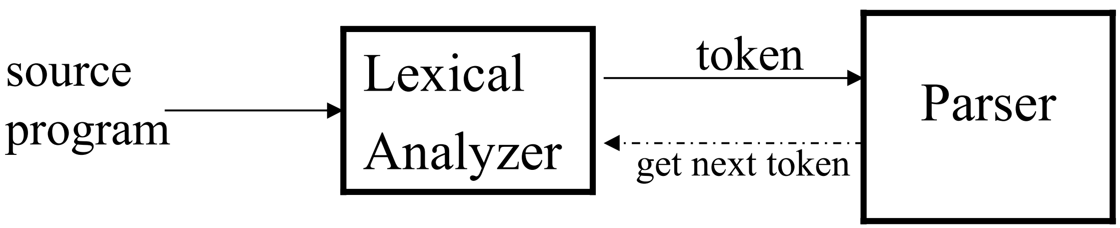
- Ex:

  (assume that we have an architecture with instructions whose at least one of its operands is a machine register)

```
MOVE    id2,R1
MULT    id3,R1
ADD     #1,R1
MOVE    R1,id1
```

Lexical Analyzer

- **Lexical Analyzer** reads the source program character by character to produce tokens.
- Normally a lexical analyzer doesn't return a list of tokens at one shot, it returns a token when the parser asks a token from it.



Token

- Token represents a set of strings described by a pattern.
  - Identifier represents a set of strings which start with a letter continues with letters and digits
  - The actual string (newval) is called as *lexeme*.
  - Tokens: identifier, number, addop, delimeter, …
- Since a token can represent more than one lexeme, additional information should be held for that specific lexeme. This additional information is called as the *attribute* of the token.
- For simplicity, a token may have a single attribute which holds the required information for that token.
  - For identifiers, this attribute a pointer to the symbol table, and the symbol table holds the actual attributes for that token.
- Some attributes:
  - <id,attr> where attr is pointer to the symbol table
  - <assgop,_> no attribute is needed (if there is only one assignment operator)
  - <num,val> where val is the actual value of the number.
- Token type and its attribute uniquely identifies a lexeme.
- ***Regular expressions*** are widely used to specify patterns.

Terminology of Languages

- **Alphabet** : a finite set of symbols (ASCII characters)
- **String** :
  - Finite sequence of symbols on an alphabet
  - Sentence and word are also used in terms of string
  - ε is the empty string
  - $|s|$ is the length of string s.
- **Language**: sets of strings over some fixed alphabet
  - $\varnothing$ the empty set is a language.
  - $\{\varepsilon\}$ the set containing empty string is a language
  - The set of well-formed C programs is a language
  - The set of all possible identifiers is a language.
- **Operators on Strings**:
  - *Concatenation*: xy represents the concatenation of strings x and y. $s\,\varepsilon = s$ $\varepsilon\, s = s$
  - $s^n = s\,s\,s\,..\,s$ (n times) $s^0 = \varepsilon$

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- Exponentiation:
  - $L^0 = \{\varepsilon\}$ $\quad L^1 = L \quad$ $L^2 = LL$
- Kleene Closure
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
- Positive Closure
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$

Example

- $L_1 = \{a,b,c,d\}$ $L_2 = \{1,2\}$

- $L_1L_2 = \{a1,a2,b1,b2,c1,c2,d1,d2\}$

- $L_1 \cup L_2 = \{a,b,c,d,1,2\}$

- L_1^3 = all strings with length three (using a,b,c,d}

- L_1^* = all strings using letters a,b,c,d and empty string

- L_1^+ = doesn't include the empty string

Regular Expressions

- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

<u>Reg. Expr</u>	<u>Language it denotes</u>
ε	$\{\varepsilon\}$
$a \in \Sigma$	$\{a\}$
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	$L(r)$

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon ,0,00,000,0000,....\}$
  - $(0|1)^* \Rightarrow$ all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in

. $\Sigma \cup \{d_1, d_2, \ldots, d_{i-1}\}$

$d_n \rightarrow r_n$

basic symbols previously defined names

Regular Definitions (cont.)

- Ex: Identifiers in Pascal

  letter $\rightarrow$ A | B | ... | Z | a | b | ... | z

  digit $\rightarrow$ 0 | 1 | ... | 9

  id $\rightarrow$ letter (letter | digit) *

  – If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

  (A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Ex: Unsigned numbers in Pascal

  digit $\rightarrow$ 0 | 1 | ... | 9

  digits $\rightarrow$ digit $^+$

  opt-fraction $\rightarrow$ (. digits) ?

  opt-exponent $\rightarrow$ (E (+|-)? digits) ?

  unsigned-num $\rightarrow$ digits opt-fraction opt-exponent

Finite Automata

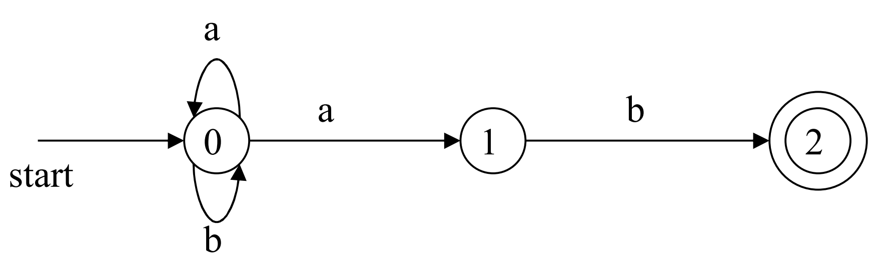
- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression ➔ NFA ➔ DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression ➔ DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - S - a set of states
  - Σ - a set of input symbols (alphabet)
  - move – a transition function move to map state-symbol pairs to sets of states.
  - s_0 - a start (initial) state
  - F – a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

NFA (Example)



Transition graph of the NFA

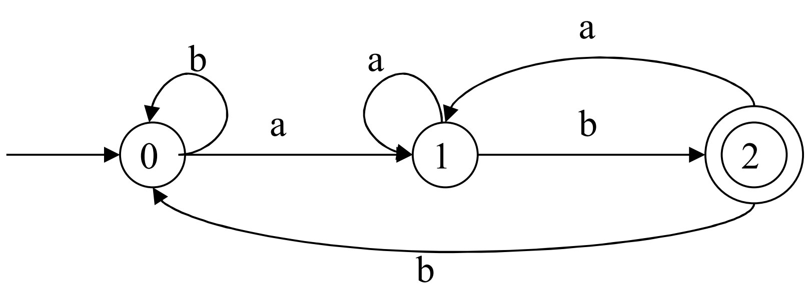
0 is the start state s_0
{2} is the set of final states F
$\Sigma = \{a,b\}$
S = {0,1,2}
Transition Function:

	a	b
0	{0,1}	{0}
1	_	{2}
2	_	_

The language recognized by this NFA is $(a|b)^*$ a b

Deterministic Finite Automaton (DFA)

- A Deterministic Finite Automaton (DFA) is a special form of a NFA.
  - no state has ε- transition
  - for each symbol a and state s, there is at most one labeled edge a leaving s.
    i.e. transition function is from pair of state-symbol to state (not set of states)



The language recognized by this DFA is also $(a|b)^*\ a\ b$

Implementing a DFA

- Let us assume that the end of a string is marked with a special symbol (say eos). The algorithm for recognition will be as follows: (an efficient implementation)

```
s ← s0                  { start from the initial state }
c ← nextchar            { get the next character from the input string }
while (c != eos) do     { do until the end of the string }
   begin
      s ← move(s,c)     { transition function }
      c ← nextchar
   end
if (s in F) then        { if s is an accepting state }
   return "yes"
else
   return "no"
```

Implementing a NFA

```
S ← ε-closure({s0})              { set all of states can be accessible from s0 by ε-transitions }
c ← nextchar
while (c != eos) {
    begin
        S ← ε-closure(move(S,c))  { set of all states can be accessible from a state in S
        c ← nextchar                by a transition on c }
    end
if (S∩F != Φ) then               { if S contains an accepting state }
    return "yes"
else
    return "no"
```

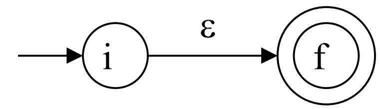
- This algorithm is not efficient.

Converting A Regular Expression into an NFA (Thomson's Construction)

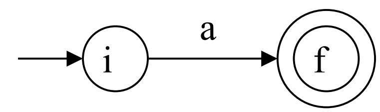
- This is one way to convert a regular expression into an NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method.
  It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
  To create an NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA

Thomson's Construction (cont.)

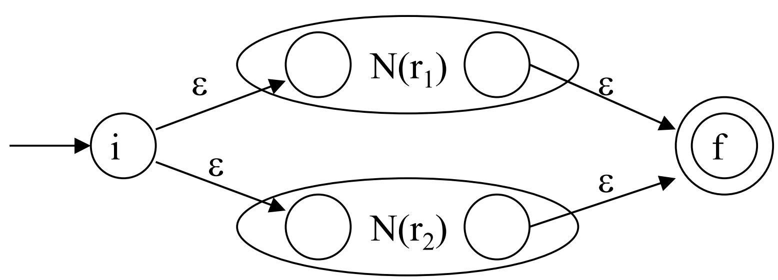
- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ



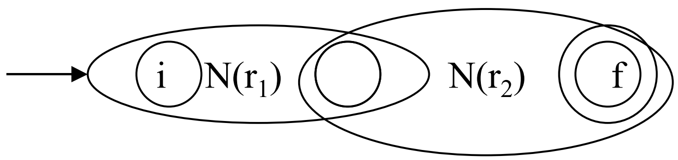
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

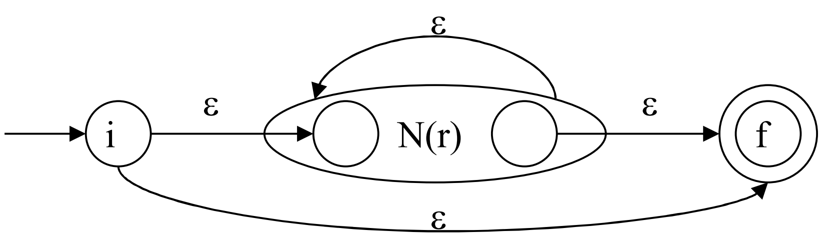
- For regular expression $r_1 \ r_2$



Final state of $N(r_2)$ become final state of $N(r_1 r_2)$

NFA for $r_1 \ r_2$

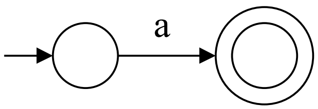
- For regular expression r^*



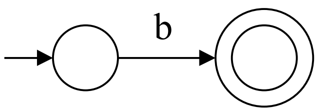
NFA for r^*

Thomson's Construction (Example - $(a|b)^*a$)

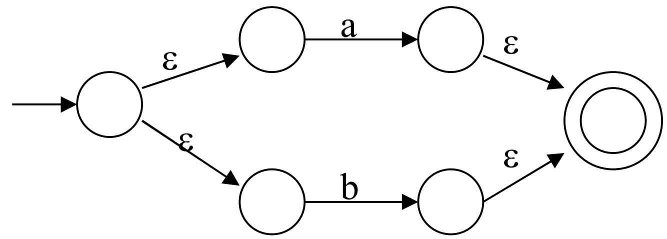
a:



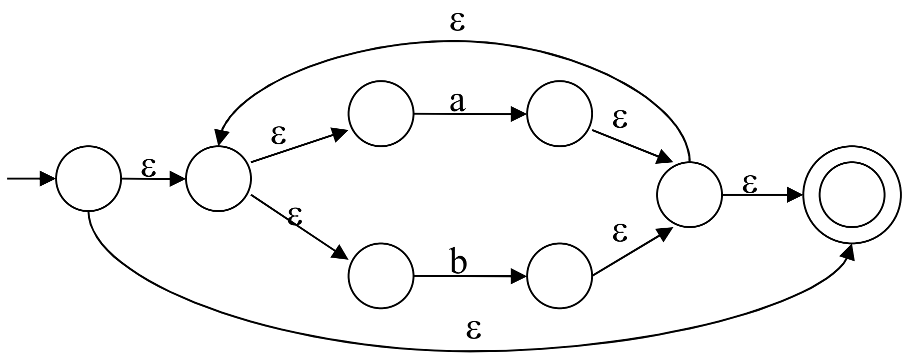
b:



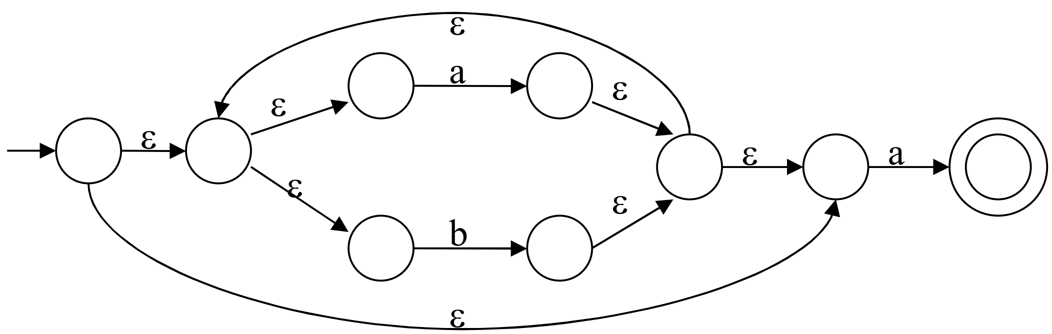
(a | b)



$(a|b)^*$



$(a|b)^*a$



Converting an NFA into a DFA (subset construction)

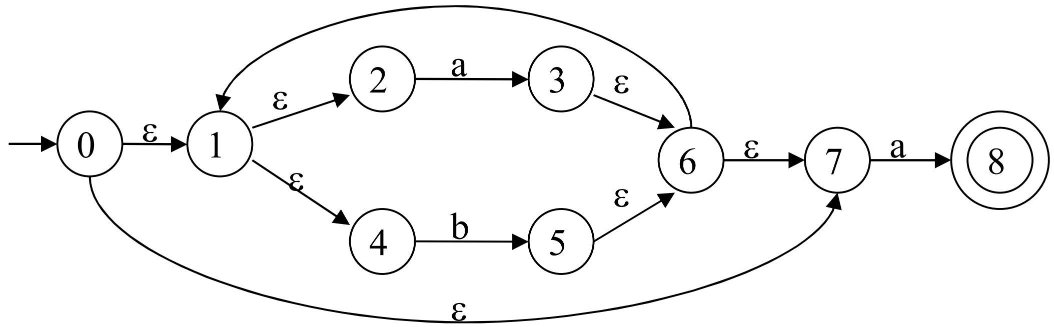
```
put  ε-closure({s0}) as an unmarked  state into the set of DFA (DS)
while (there is one unmarked S1 in DS) do
    begin
        mark S1
        for each input symbol a do
            begin
                S2 ← ε-closure(move(S1,a))
                if (S2 is not in DS) then
                    add S2 into DS as an unmarked state
                transfunc[S1,a] ← S2
            end
    end
```

ε-closure($\{s_0\}$) is the set of all states can be accessible from s_0 by ε-transition.

set of states to which there is a transition on a from a state s in S_1

- a state S in DS is an accepting state of DFA if a state in S is an accepting state of NFA
- the start state of DFA is ε-closure($\{s_0\}$)

Converting an NFA into a DFA (Example)



S_0 = ε-closure({0}) = {0,1,2,4,7} S_0 into DS as an unmarked state

⇓ mark S_0

ε-closure(move(S_0,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1 S_1 into DS

ε-closure(move(S_0,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2 S_2 into DS

transfunc[S_0,a] ⇐ S_1 transfunc[S_0,b] ⇐ S_2

⇓ mark S_1

ε-closure(move(S_1,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_1,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_1,a] ⇐ S_1 transfunc[S_1,b] ⇐ S_2

⇓ mark S_2

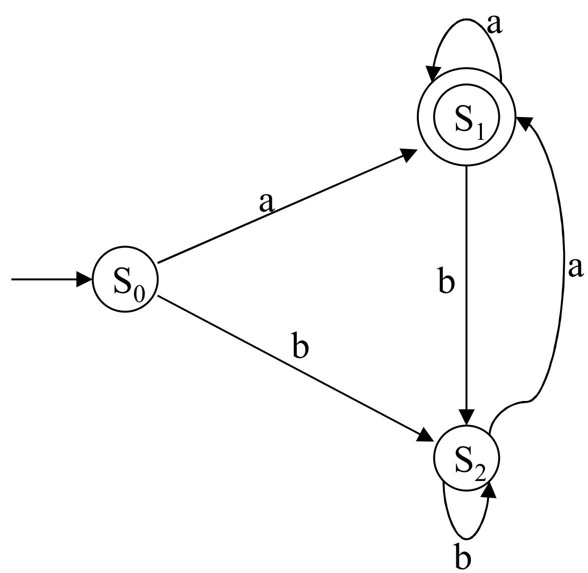
ε-closure(move(S_2,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = S_1

ε-closure(move(S_2,b)) = ε-closure({5}) = {1,2,4,5,6,7} = S_2

transfunc[S_2,a] ⇐ S_1 transfunc[S_2,b] ⇐ S_2

Converting an NFA into a DFA (Example – cont.)

S_0 is the start state of DFA since 0 is a member of $S_0=\{0,1,2,4,7\}$
S_1 is an accepting state of DFA since 8 is a member of $S_1 = \{1,2,3,4,6,7,8\}$



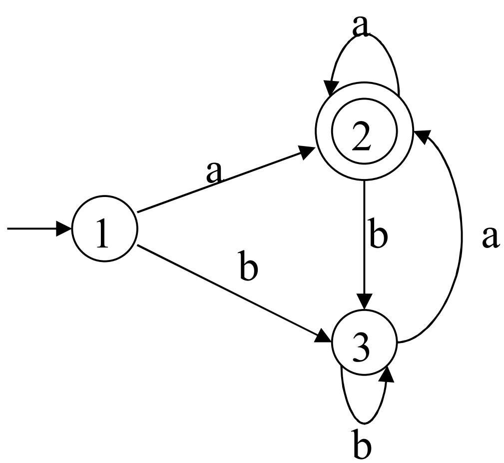
Minimizing Number of States of a DFA

- partition the set of states into two groups:
  - G_1 : set of accepting states
  - G_2 : set of non-accepting states

- For each new group G
  - partition G into subgroups such that states s_1 and s_2 are in the same group iff for all input symbols a, states s_1 and s_2 have transitions to states in the same group.

- Start state of the minimized DFA is the group containing the start state of the original DFA.
- Accepting states of the minimized DFA are the groups containing the accepting states of the original DFA.

Minimizing DFA - Example

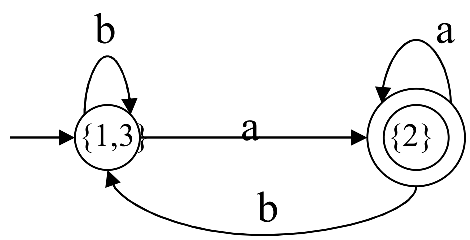


$G_1 = \{2\}$
$G_2 = \{1,3\}$

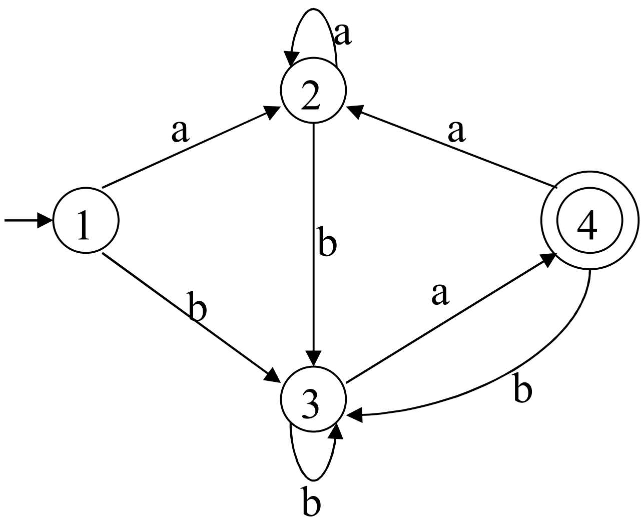
G_2 cannot be partitioned because

move(1,a)=2   move(1,b)=3
move(3,a)=2   move(3,b)=3

So, the minimized DFA (with minimum states)



Minimizing DFA – Another Example



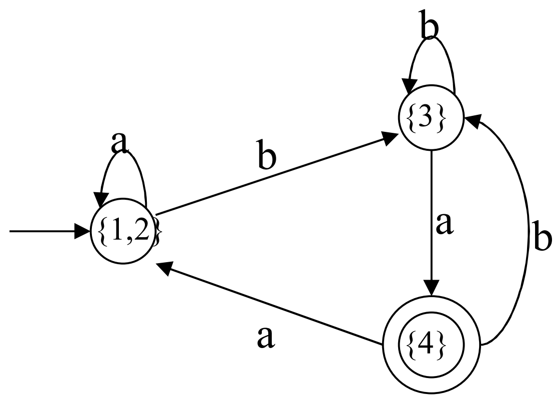
Groups: {1,2,3} {4}

{1,2} {3}

no more partitioning

a	b
1->2	1->3
2->2	2->3
3->4	3->3

So, the minimized DFA



Some Other Issues in Lexical Analyzer

- The lexical analyzer must recognize the longest possible string.
  - Ex: identifier `newval` -- `n` `ne` `new` `newv` `newva` `newval`

- What is the end of a token? Is there any character which marks the end of a token?
  - It is normally not defined.
  - If the number of characters in a token is fixed, in that case no problem: + -
  - But < ➔ < or <> (in Pascal)
  - The end of an identifier : the characters cannot be in an identifier can mark the end of token.
  - We may need a lookhead
    - In Prolog: p :- X is 1. p :- X is 1.5.
      The dot followed by a white space character can mark the end of a number.
      But if that is not the case, the dot must be treated as a part of the number.

Some Other Issues in Lexical Analyzer (cont.)

- Skipping comments
  - Normally we don’t return a comment as a token.
  - We skip a comment and return the next token (which is not a comment) to the parser.
  - So, the comments are only processed by the lexical analyzer, and the don’t complicate the syntax of the language.

- Symbol table interface
  - symbol table holds information about tokens (at least lexeme of identifiers)
  - how to implement the symbol table, and what kind of operations.
    - hash table – open addressing, chaining
    - putting into the hash table, finding the position of a token from its lexeme.

- Positions of the tokens in the file (for the error handling).