CS 540 Spring 2013

The Course covers:

- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Runtime environments
- Code Generation
- Code Optimization

Pre-requisite courses

- **Strong** programming background in C, C++ or Java – CS 310
- Formal Language (NFAs, DFAs, CFG) – CS 330
- Assembly Language Programming and Machine Architecture –CS 367

Operational Information

- Office: Engineering Building, Rm. 5315
- E-mail: white@gmu.edu
- Class Web Page: Blackboard
- Discussion board: Piassa
- Computer Accounts on `zeus.vse.gmu.edu` (link on 'Useful Links')

CS 540 Course Grading

- Programming Assignments (45%)
  - 5% + 10% + 10% + 20%
- Exams – midterm and final (25%, 30%)

Resources

- Textbooks:
  - *Compilers: Principles*, *Techniques and Tools*, Aho, Lam, Sethi & Ullman, 2007 (required)
  - *lex & yacc*, Levine et. al.
- Slides
- Sample code for Lex/YACC (C, C++, Java)

Distance Education

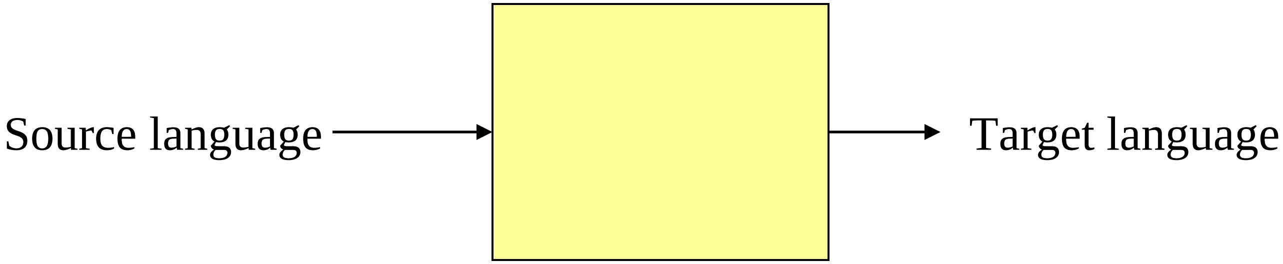
- CS 540 Spring ‘13 session is delivered to the Internet section (Section 540-DL) online by NEW
- Students in distance section will access to online lectures and can play back the lectures and download the PDF slide files
- The distance education students will be given the midterm and final exam on campus, on the same day/time as in class students. Exam locations will be announced closer to the exam dates.

Lecture 1: Introduction to Language Processing & Lexical Analysis

CS 540

What is a compiler?

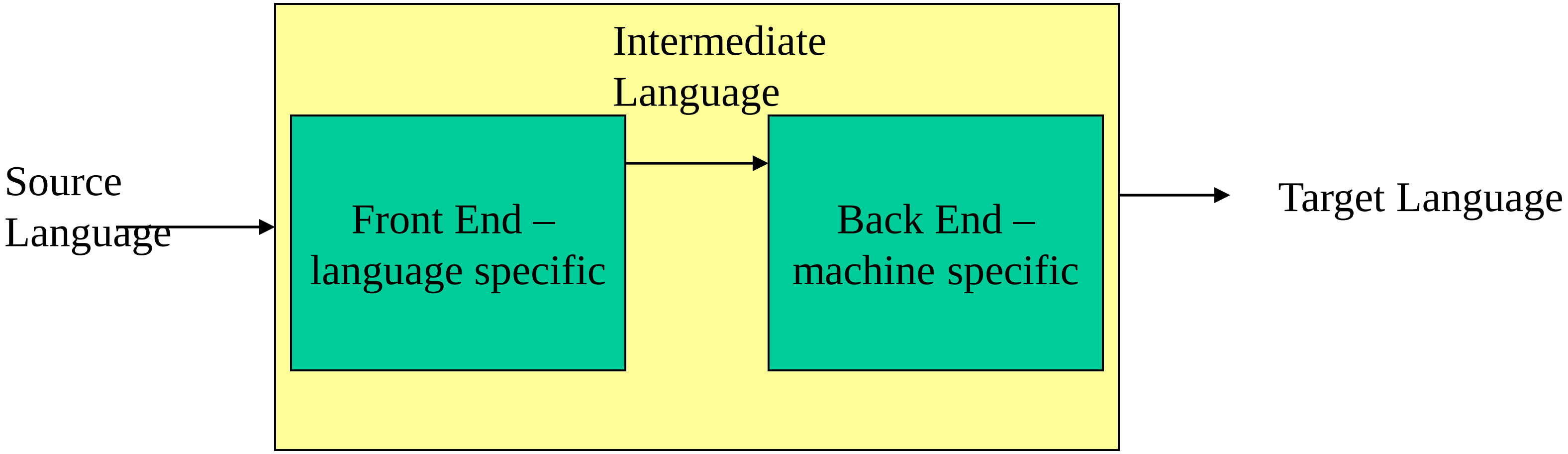
A program that reads a program written in one language and translates it into another language.



Traditionally, compilers go from high-level languages to low-level languages.

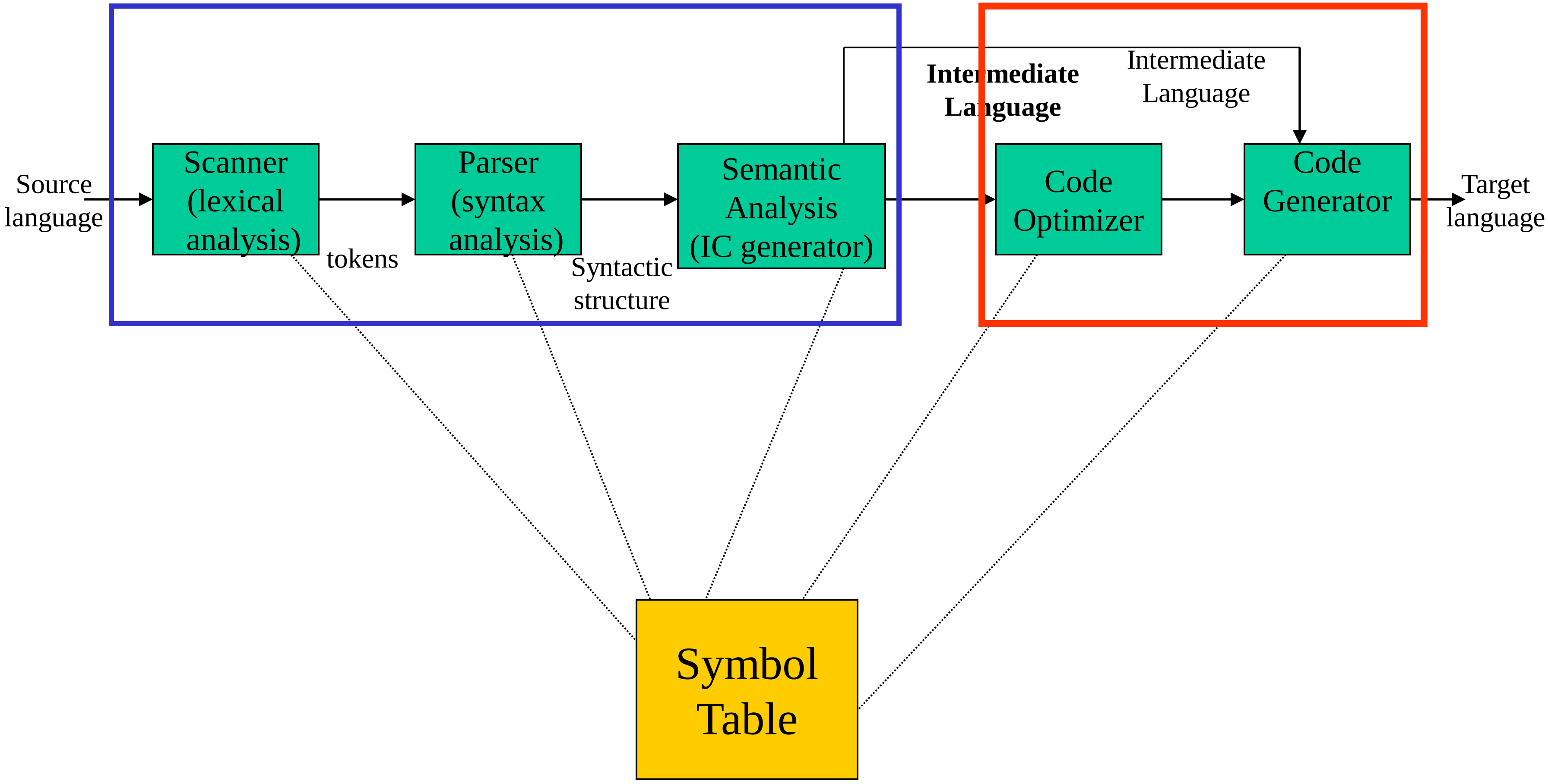
Compiler Architecture

In more detail:

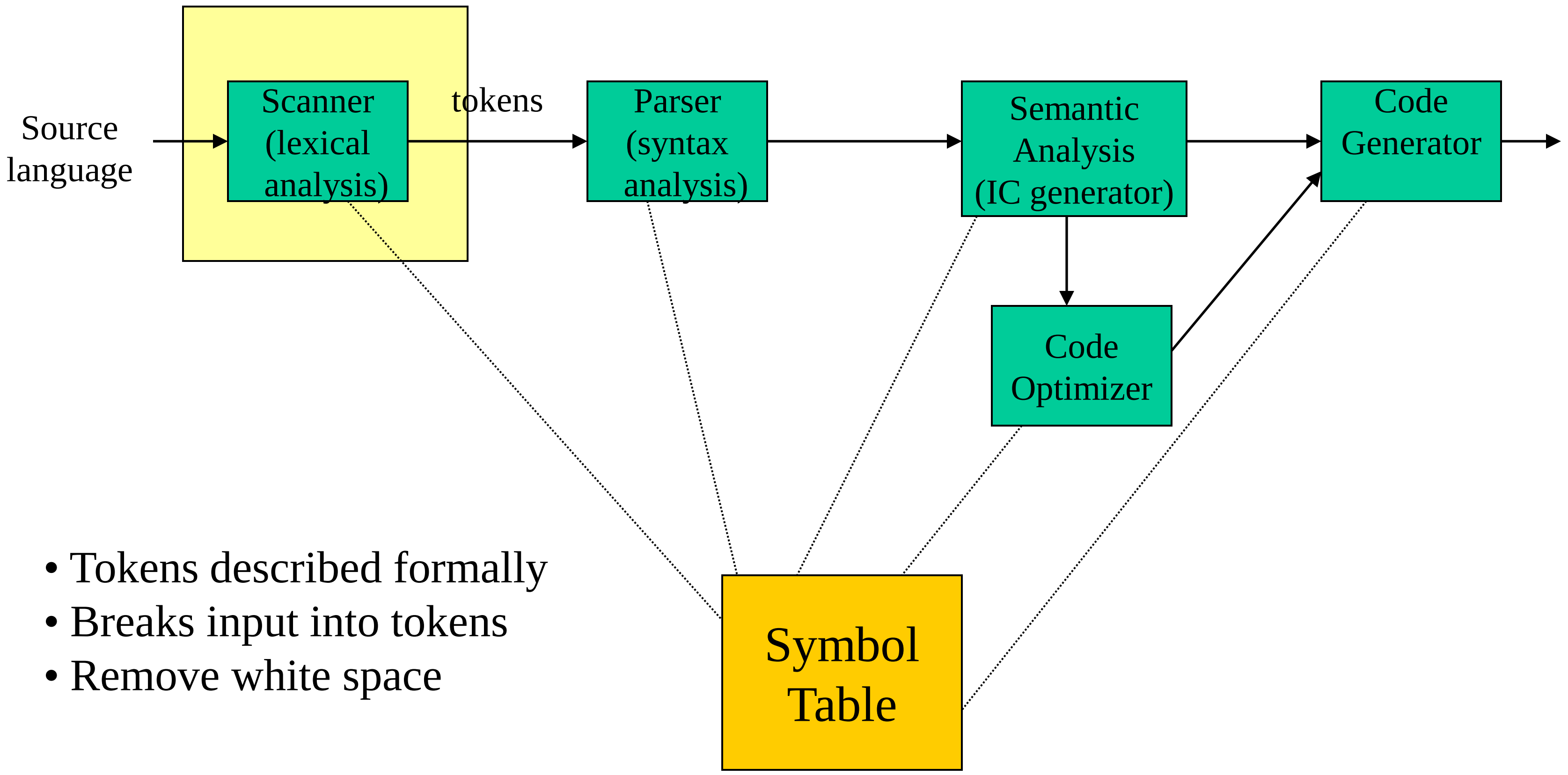


- Separation of Concerns
- Retargeting

Compiler Architecture



Lexical Analysis - Scanning

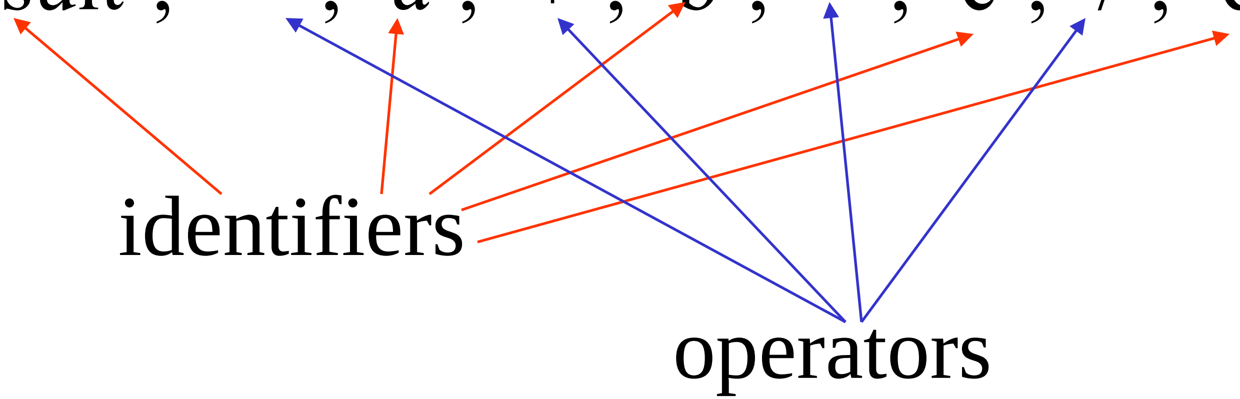


- Tokens described formally
- Breaks input into tokens
- Remove white space

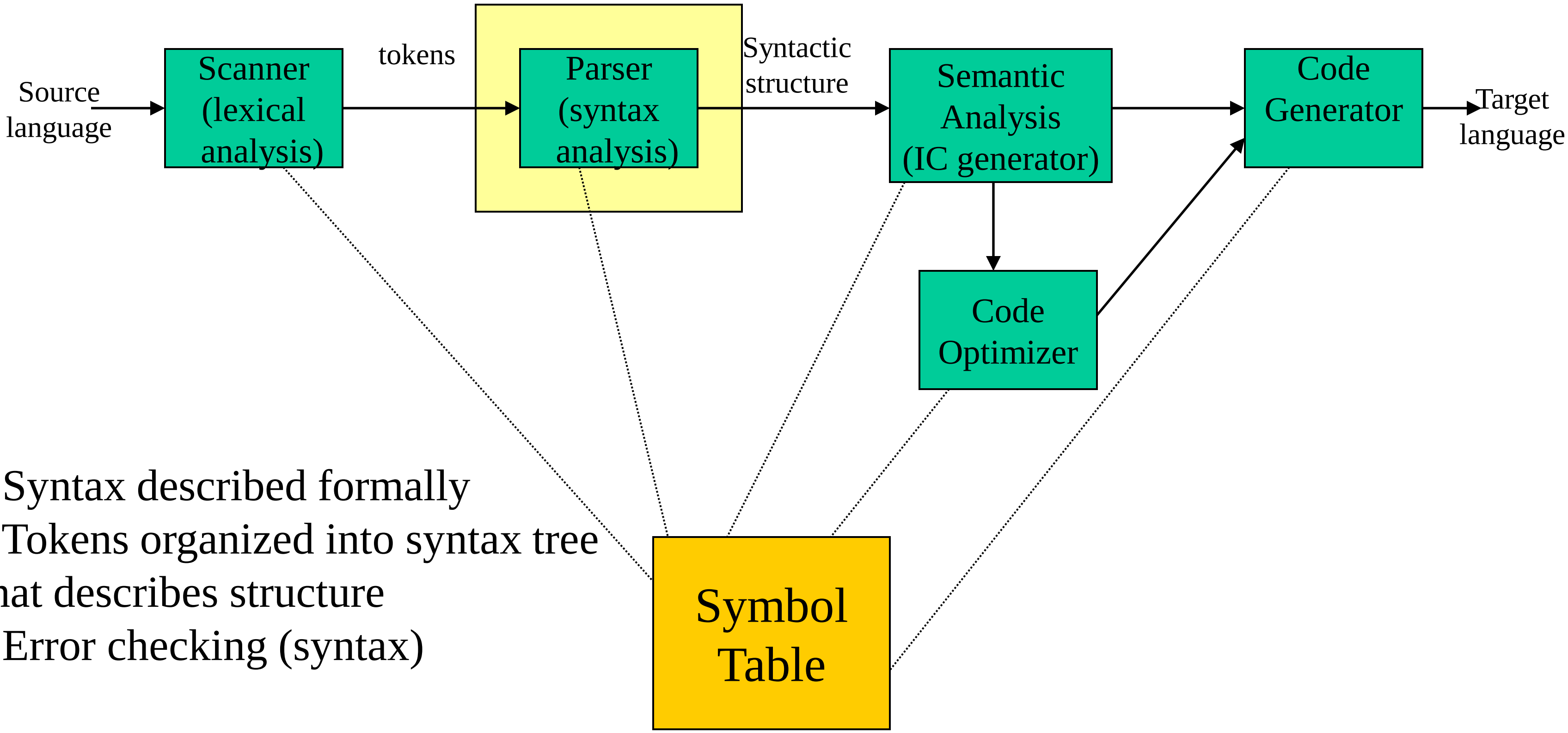
Input: result = a + b * c / d

- Tokens:

‘result’, ‘=‘, ‘a’, ‘+’, ‘b’, ‘*’, ‘c’, ‘/’, ‘d’



Static Analysis - Parsing

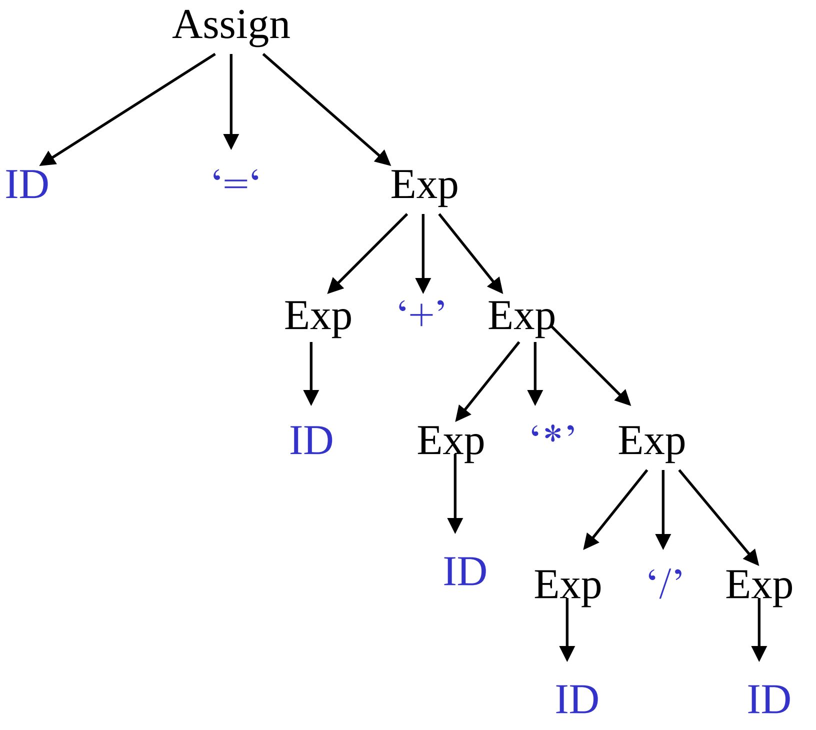


- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking (syntax)

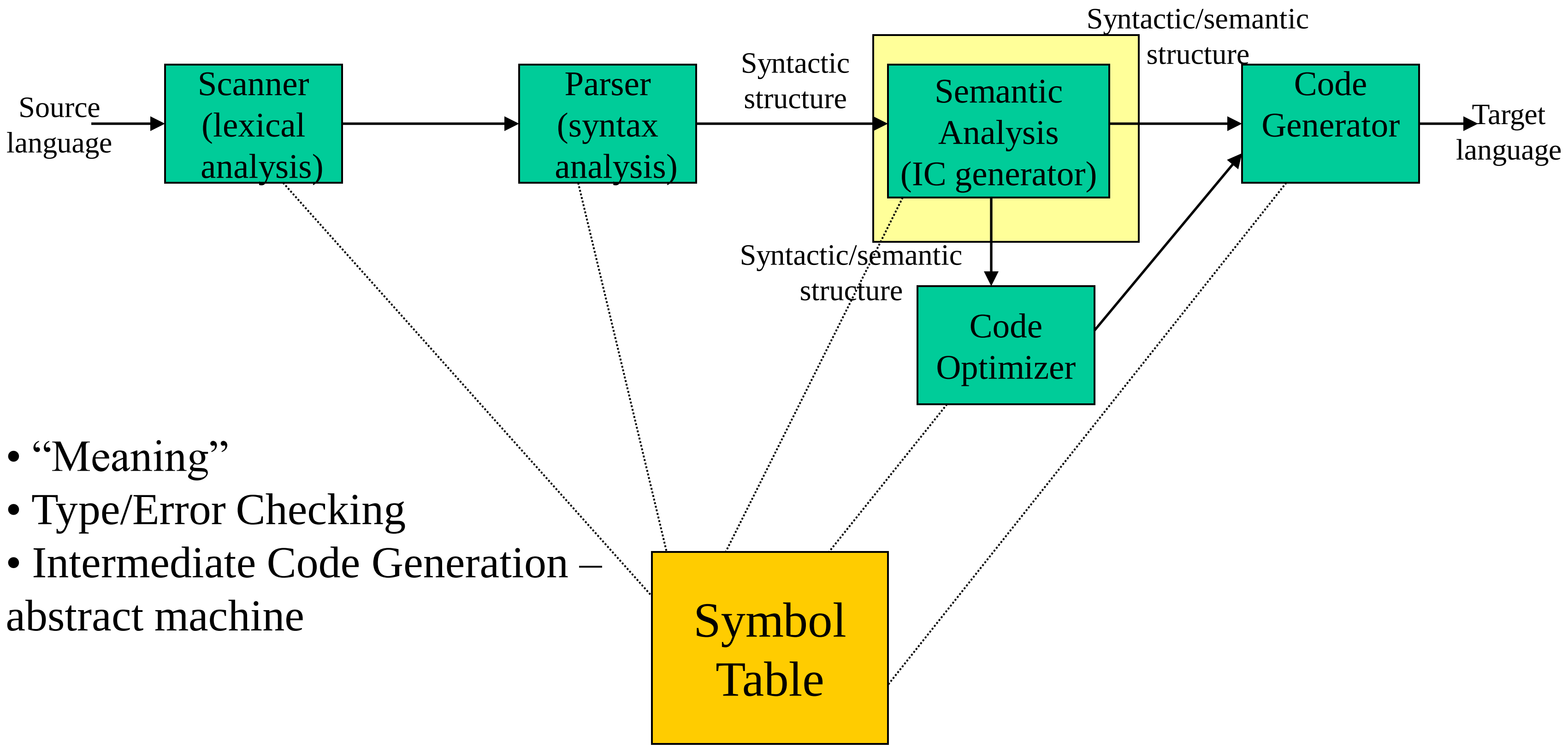
Input: result = a + b * c / d

Exp ::= Exp '+' Exp
| Exp '-' Exp
| Exp '*' Exp
| Exp '/' Exp
| ID

Assign ::= ID '=' Exp

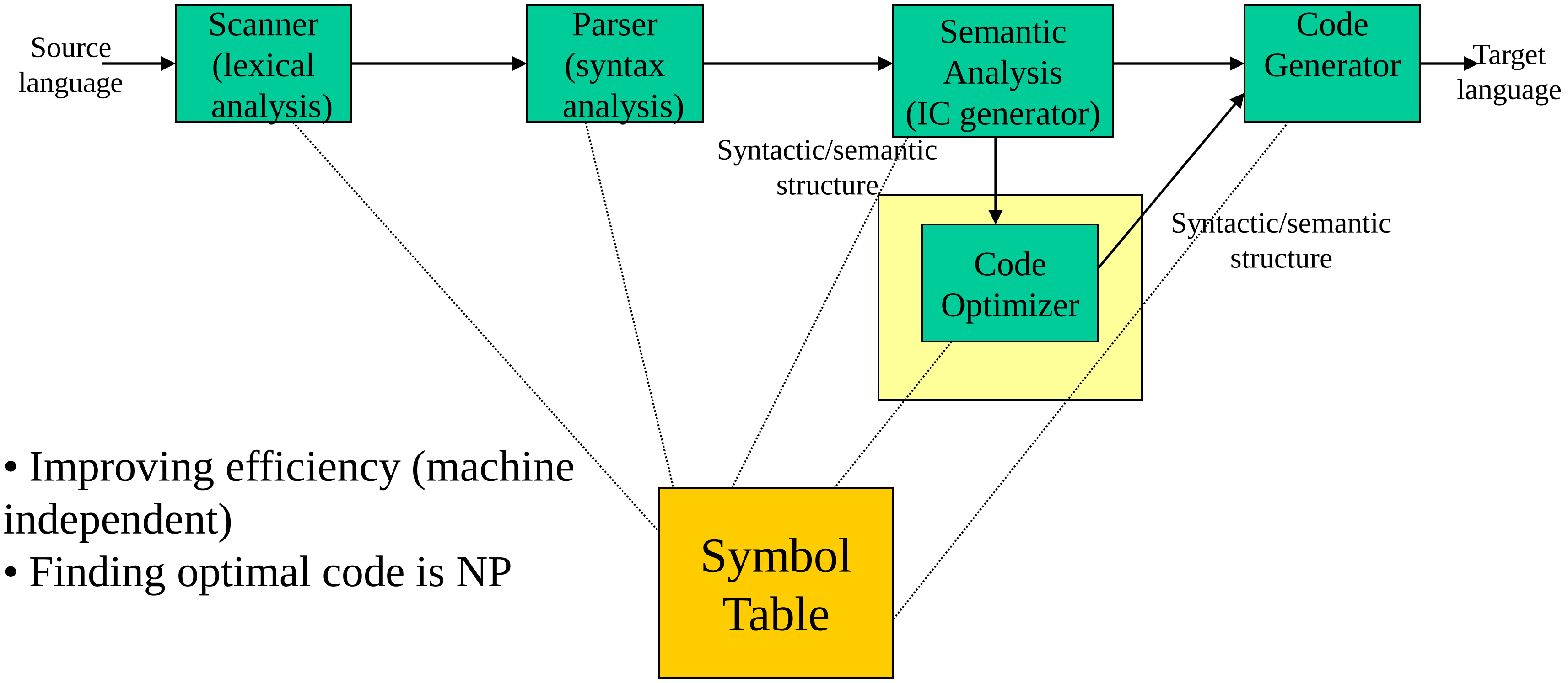


Semantic Analysis



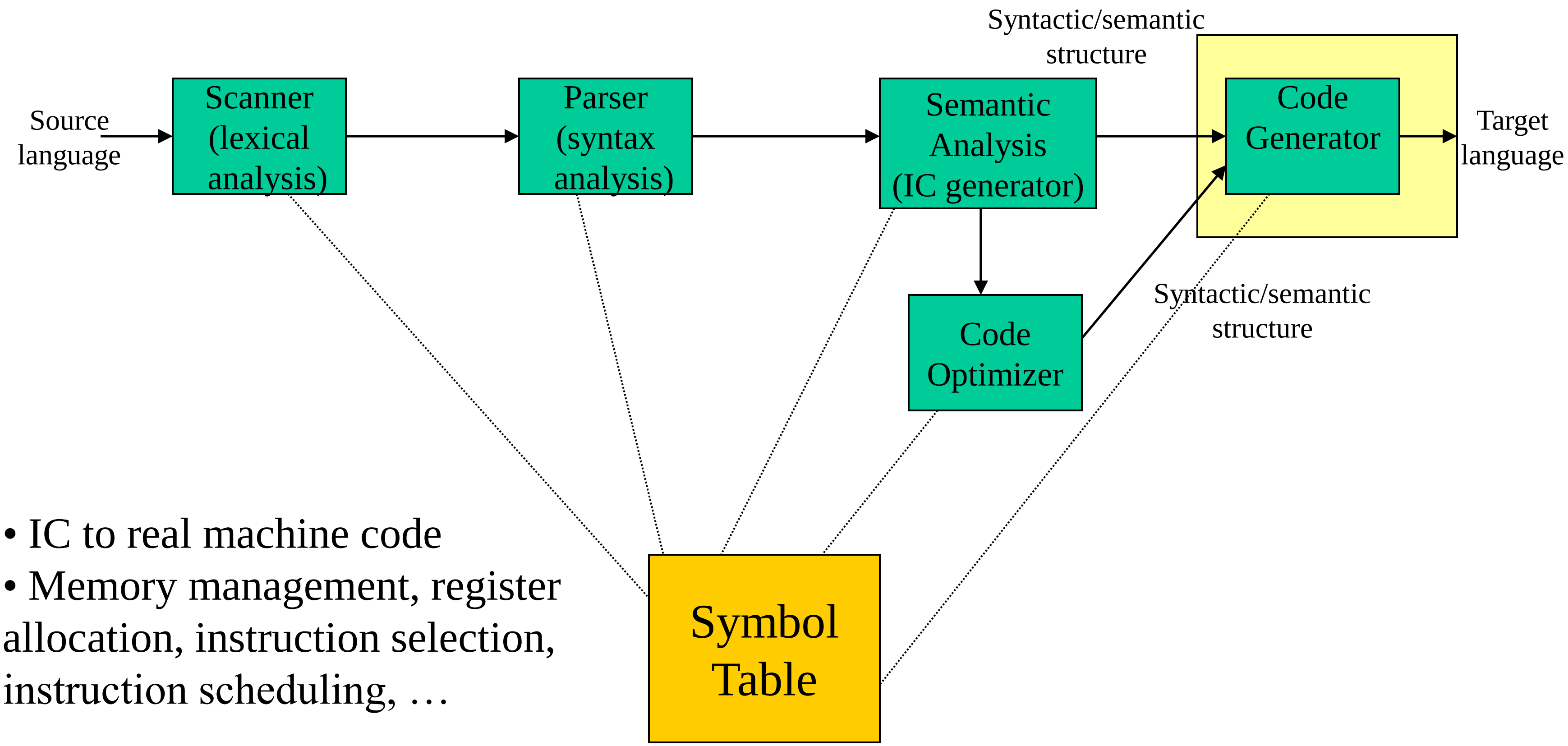
- "Meaning"
- Type/Error Checking
- Intermediate Code Generation – abstract machine

Optimization



- Improving efficiency (machine independent)
- Finding optimal code is NP

Code Generation



- IC to real machine code
- Memory management, register allocation, instruction selection, instruction scheduling, …

Issues Driving Compiler Design

- Correctness
- Speed (runtime and compile time)
  - Degrees of optimization
  - Multiple passes
- Space
- Feedback to user
- Debugging

Related to Compilers

- Interpreters (direct execution)
- Assemblers
- Preprocessors
- Text formatters (non-WYSIWYG)
- Analysis tools

Why study compilers?

- Bring together:
  - Data structures & Algorithms
  - Formal Languages
  - Computer Architecture
- Influence:
  - Language Design
  - Architecture (influence is bi-directional)
- Techniques used influence other areas (program analysis, testing, …)

Review of Formal Languages

- Regular expressions, NFA, DFA
- Translating between formalisms
- Using these formalisms

What is a language?

- **Alphabet** – finite character set (Σ)
- **String** – finite sequence of characters – can be ε, the empty string (Some texts use λ as the empty string)
- **Language** – possibly infinite set of strings over some alphabet – can be { }, the empty language.

Suppose Σ = {a,b,c}. Some languages over Σ could be:

- {aa,ab,ac,bb,bc,cc}
- {ab,abc,abcc,abccc,. . .}
- { ε }
- { }
- {a,b,c,ε}
- …

Why do we care about Regular Languages?

- Formally describe tokens in the language
  - Regular Expressions
  - NFA
  - DFA
- Regular Expressions ➔ finite automata
- Tools assist in the process

Regular Expressions

The **regular expressions** over finite Σ are the strings over the alphabet Σ + {), (, |, * } such that:

1. { } (empty set) is a regular expression for the empty set
2. ε is a regular expression denoting { ε }
3. *a* is a regular expression denoting set { *a* } for any *a* in Σ

Regular Expressions

4. If P and Q are regular expressions over Σ, then so are:

- **P | Q (<u>union</u>)**

  If P denotes the set {*a*,…,*e*}, Q denotes the set {*0*,…,*9*} then P | Q denotes the set {*a*,…,*e*,*0*,…,*9*}

- **PQ (<u>concatenation</u>)**

  If P denotes the set {*a*,…,*e*}, Q denotes the set {*0*,…,*9*} then PQ denotes the set {*a0*,…,*e0*,*a1*,…,*e9*}

- **Q* (<u>closure</u>)**

  If Q denotes the set {*0*,…,*9*} then Q* denotes the set {ε,*0*,…,*9*,*00*,…*99*,…}

Examples

If $\Sigma = \{a,b\}$

- (a | b)(a | b)
- (a | b)*b
- a*b*a*
- a*a (also known as a+)
- (ab*)|(a*b)

Nondeterministic Finite Automata

A **nondeterministic finite automaton** (NFA) is a mathematical model that consists of

1. A set of states S
2. A set of input symbols Σ
3. A transition function that maps state/symbol pairs to a set of states:

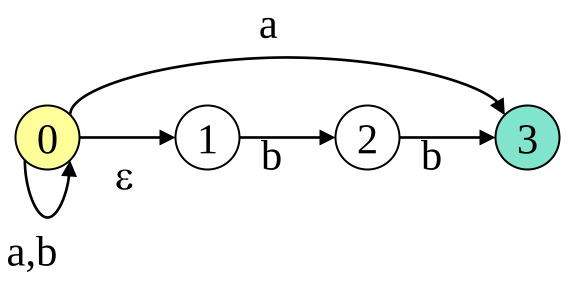
   S x {Σ + ε} → set of S

4. A special state s_0 called the start state
5. A set of states F (subset of S) of final states

INPUT: string

OUTPUT: yes or no

Example NFA



Transition Table:

STATE	a	b	ε
0	0,3	0	1
1		2	
2		3	
3			

$S = \{0,1,2,3\}$

$S_0 = 0$

$\Sigma = \{a,b\}$

$F = \{3\}$

NFA Execution

An NFA says 'yes' for an input string if there is some path from the start state to some final state where all input has been processed.

```
NFA(int s0, int input) {
   if (all input processed && s0 is a final state) return Yes;
   if (all input processed && s0 not a final state) return No;

   for all states s1 where transition(s0,table[input]) = s1
        if (NFA(s1,input_element+1) == Yes) return Yes;

   for all states s1 where transition(s0,ε) = s1
        if (NFA(s1,input_element) == Yes) return Yes;
    return No;
}
```

Uses backtracking to search all possible paths

Deterministic Finite Automata

A **deterministic finite automaton** (DFA) is a mathematical model that consists of

1. A set of states S
2. A set of input symbols Σ
3. A transition function that maps state/symbol pairs to a state:

   $S \times \Sigma \rightarrow S$
4. A special state s_0 called the start state
5. A set of states F (subset of S) of final states

INPUT: string

OUTPUT: yes or no

DFA Execution

```
DFA(int start_state) {
   state current = start_state;
   input_element = next_token();
   while (input to be processed) {
       current =
              transition(current,table[input_element])
       if current is an error state return No;
       input_element = next_token();
    }
    if current is a final state return Yes;
    else return No;
}
```

Regular Languages

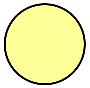
1. There is an algorithm for converting any RE into an NFA.
2. There is an algorithm for converting any NFA to a DFA.
3. There is an algorithm for converting any DFA to a RE.

These facts tell us that REs, NFAs and DFAs have equivalent expressive power. All three describe the class of regular languages.

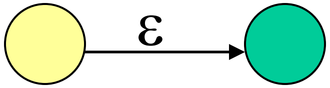
Converting Regular Expressions to NFAs

The **regular expressions** over finite Σ are the strings over the alphabet Σ + {), (, |, *} such that:

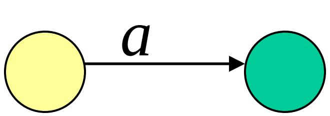
- { } (empty set) is a regular expression for the empty set



- Empty string ε is a regular expression denoting { ε }



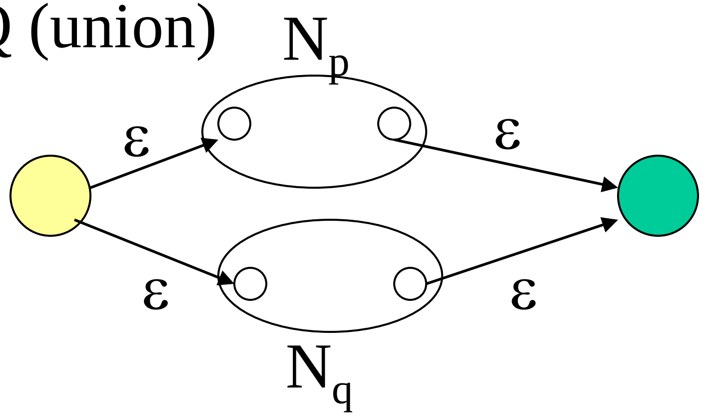
- *a* is a regular expression denoting {*a* } for any *a* in Σ



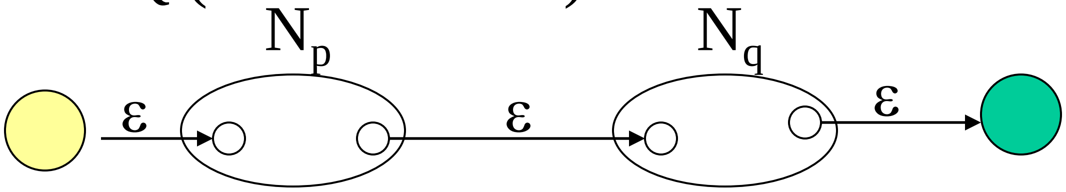
Converting Regular Expressions to NFAs

If P and Q are regular expressions with NFAs N_p, N_q:

P | Q (union)



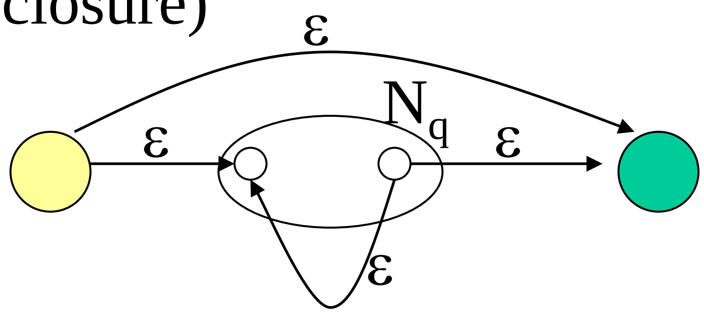
PQ (concatenation)



Converting Regular Expressions to NFAs

If Q is a regular expression with NFA N_q:

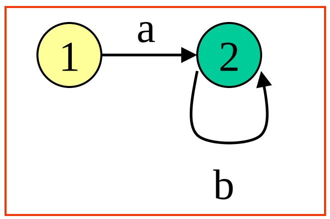
Q* (closure)



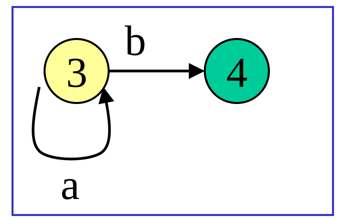
Example (ab* | a*b)*

Starting with:

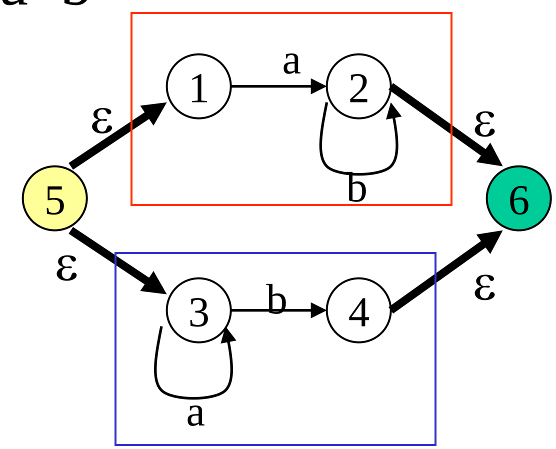
ab*



a*b

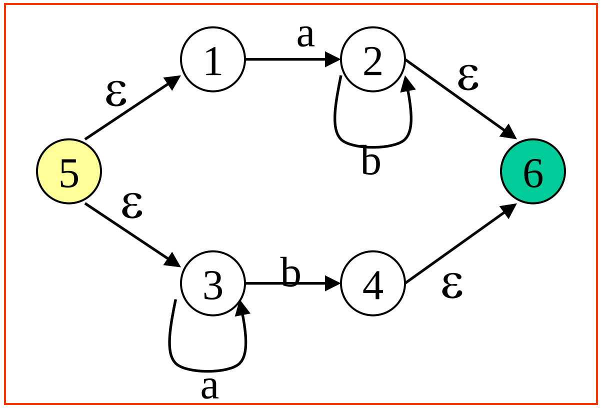


ab* | a*b

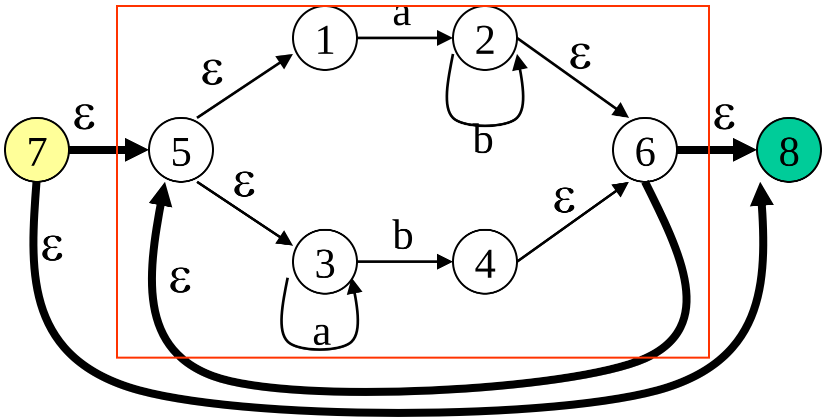


Example (ab* | a*b)*

ab* | a*b



(ab* | a*b)*



Converting NFAs to DFAs

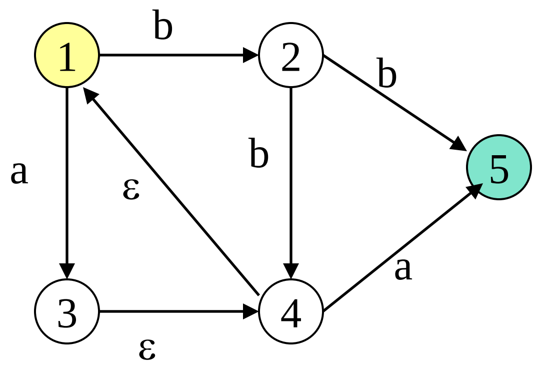
- **Idea**: Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0,s_1,\ldots\}$ after input if the NFA could be in *any* of these states for the same input.

- **Input**: NFA N with state set S_N, alphabet Σ, start state s_N, final states F_N, transition function T_N: $S_N \times \Sigma + \{\varepsilon\} \rightarrow$ set of S_N
- **Output**: DFA D with state set S_D, alphabet Σ, start state

$s_D = \varepsilon\text{-closure}(s_N)$, final states F_D, transition function T_D: $S_D \times \Sigma \rightarrow S_D$

ε-closure()

Defn: ε-closure(T) = T + all NFA states reachable from any state in T using only ε transitions.



ε-closure({1,2,5}) = {1,2,5}
ε-closure({4}) = {1,4}
ε-closure({3}) = {1,3,4}
ε-closure({3,5}) = {1,3,4,5}

Algorithm: Subset Construction

s_D = ε-closure(s_N) -- create start state for DFA

S_D = {s_D} (unmarked)

while there is some unmarked state **R** in S_D

    mark state **R**

    for all *a* in Σ do

        s = ε-closure(T_N(**R**,*a*));

        if s not already in S_D then add it (unmarked)

        T_D(**R**,*a*) = s;

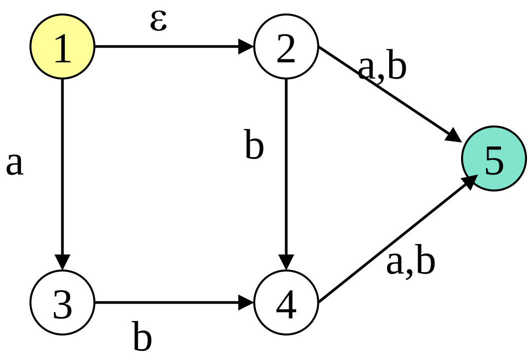
    end for

end while

F_D = any element of S_D that contains a state in F_N

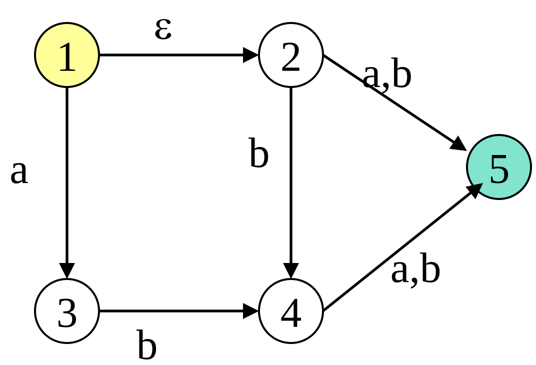
Example 1: Subset Construction

NFA



Example 1: Subset Construction

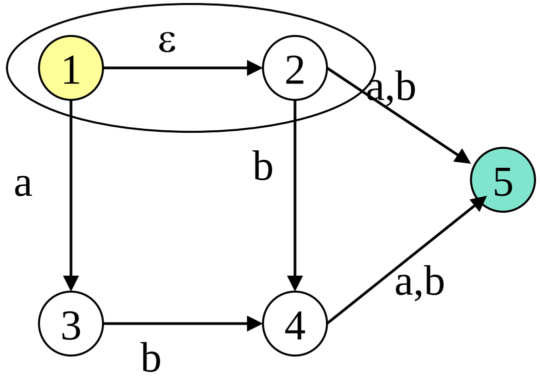
NFA

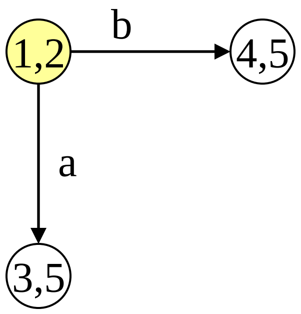


	a	b
{1,2}		

Example 1: Subset Construction

NFA

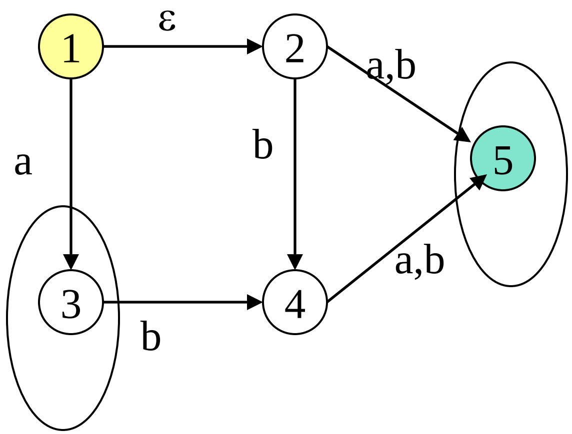


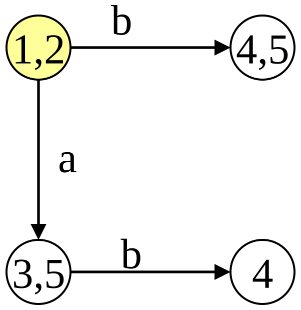


	a	b
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 1: Subset Construction

NFA

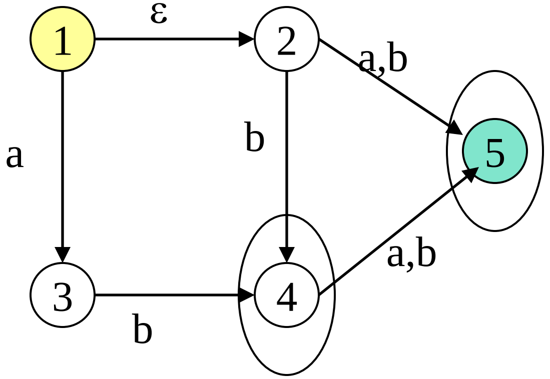


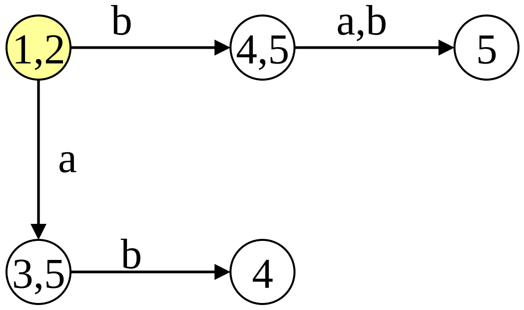


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 1: Subset Construction

NFA

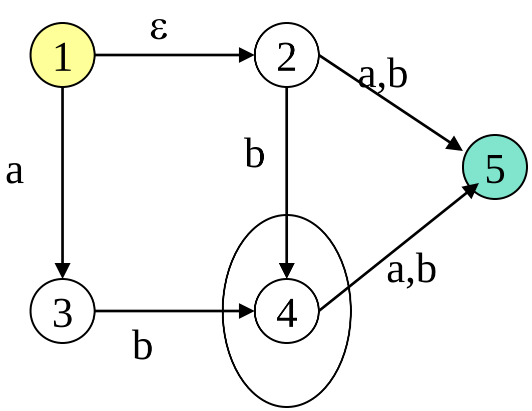


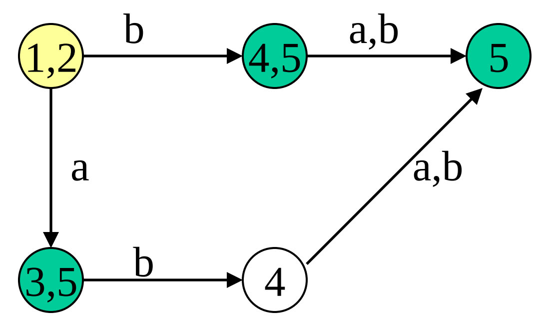


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 1: Subset Construction

NFA



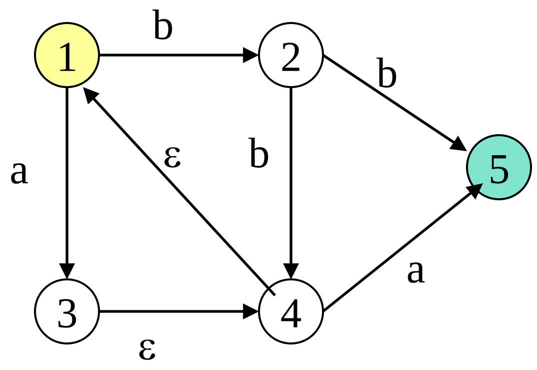


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

All final states since the NFA final state is included

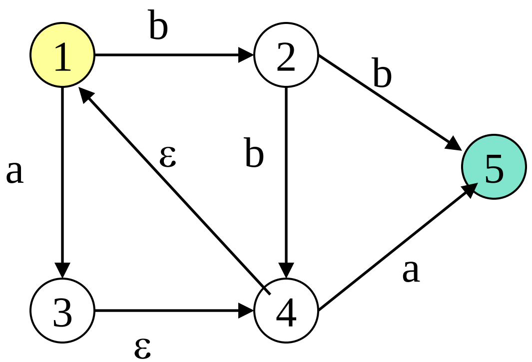
Example 2: Subset Construction

NFA

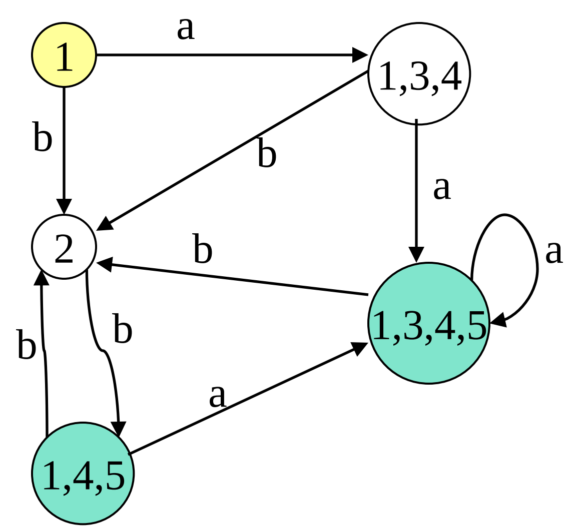


Example 2: Subset Construction

NFA

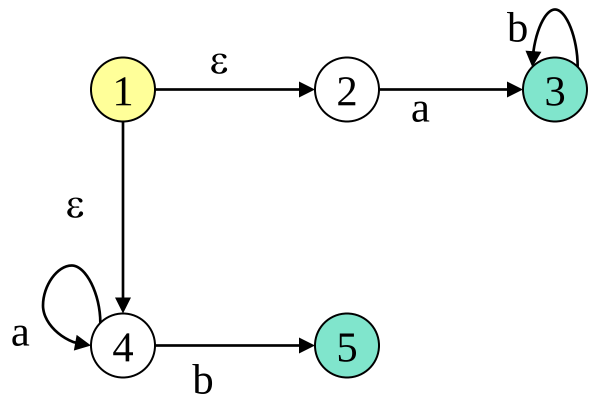


DFA

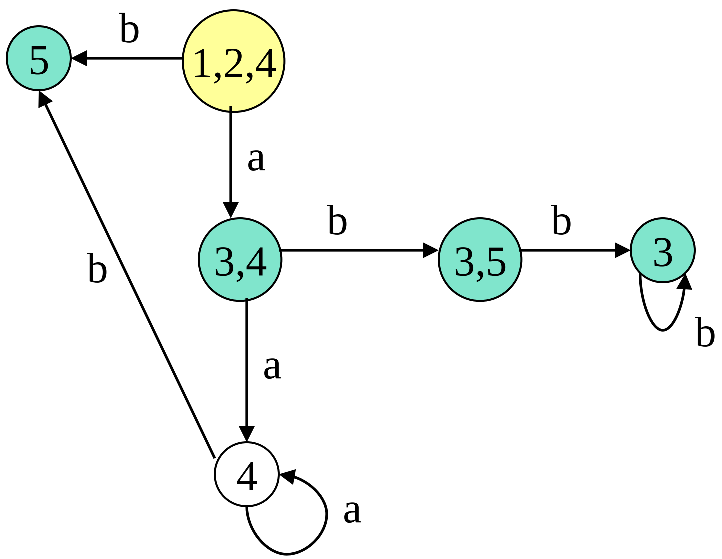


Example 3: Subset Construction

NFA



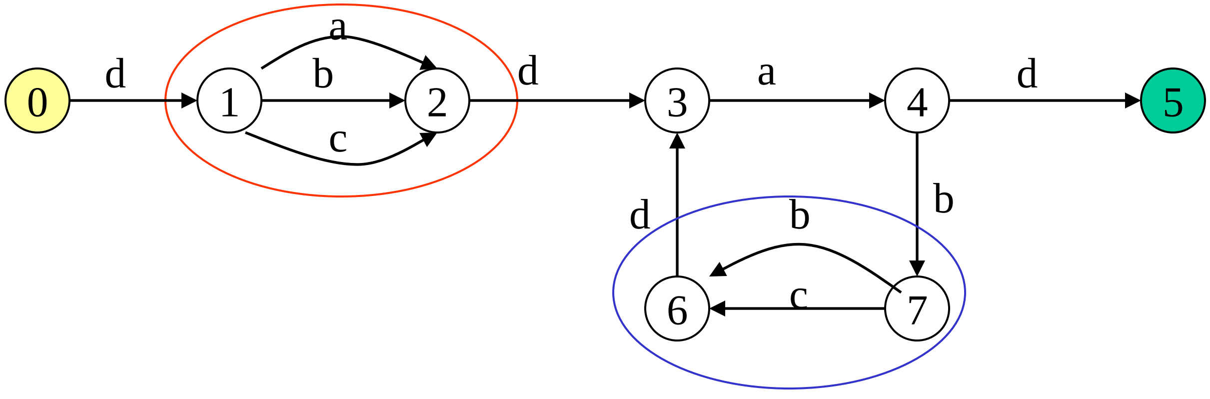
DFA



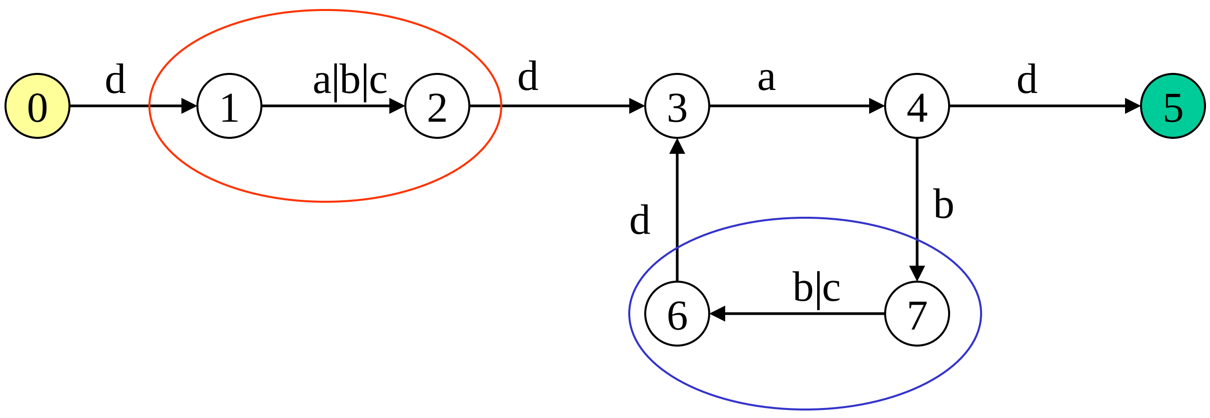
Converting DFAs to REs

1. Combine serial links by concatenation
2. Combine parallel links by alternation
3. Remove self-loops by Kleene closure
4. Select a node (other than initial or final) for removal. Replace it with a set of equivalent links whose path expressions correspond to the in and out links
5. Repeat steps 1-4 until the graph consists of a single link between the entry and exit nodes.

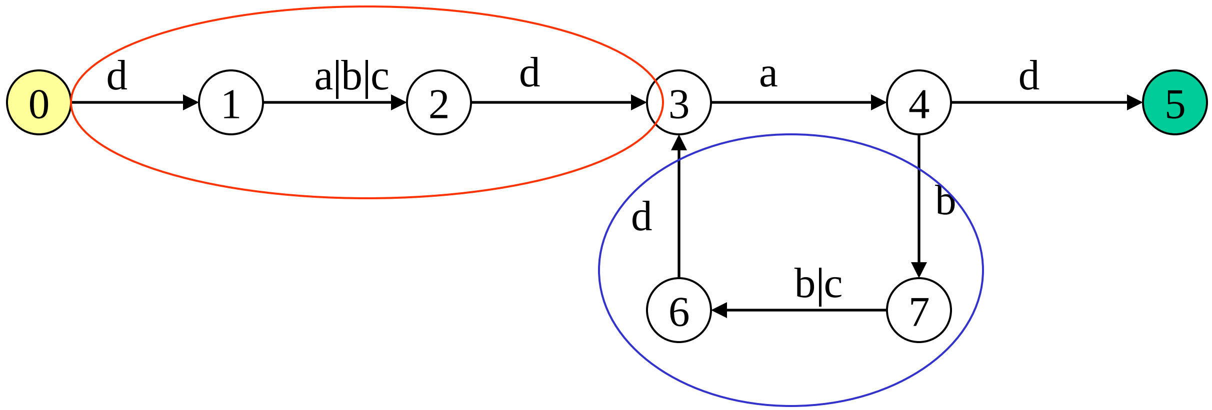
Example



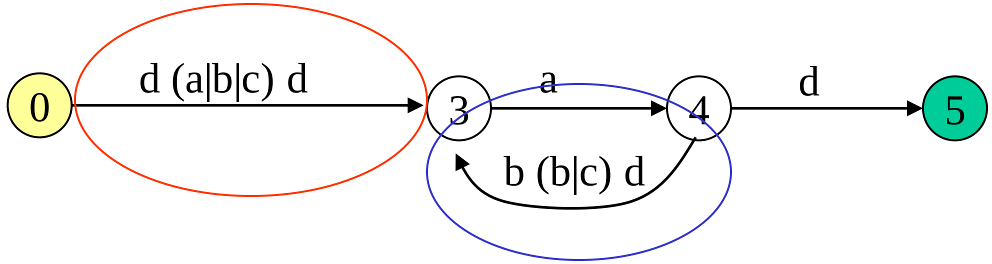
parallel edges become alternation



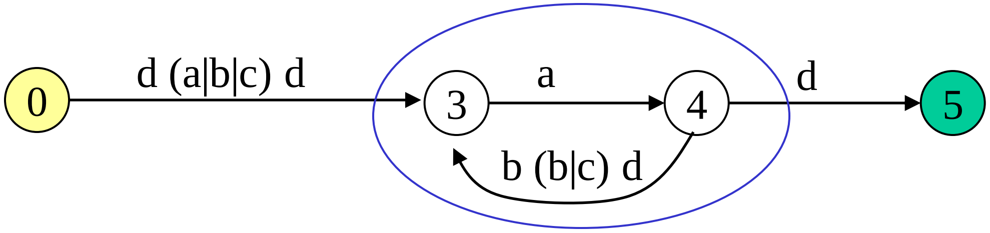
Example



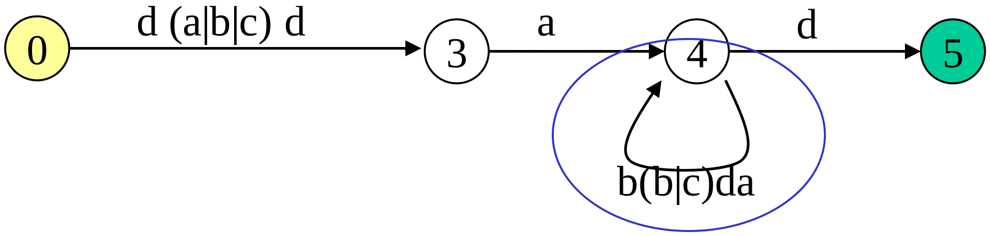
serial edges become concatenation



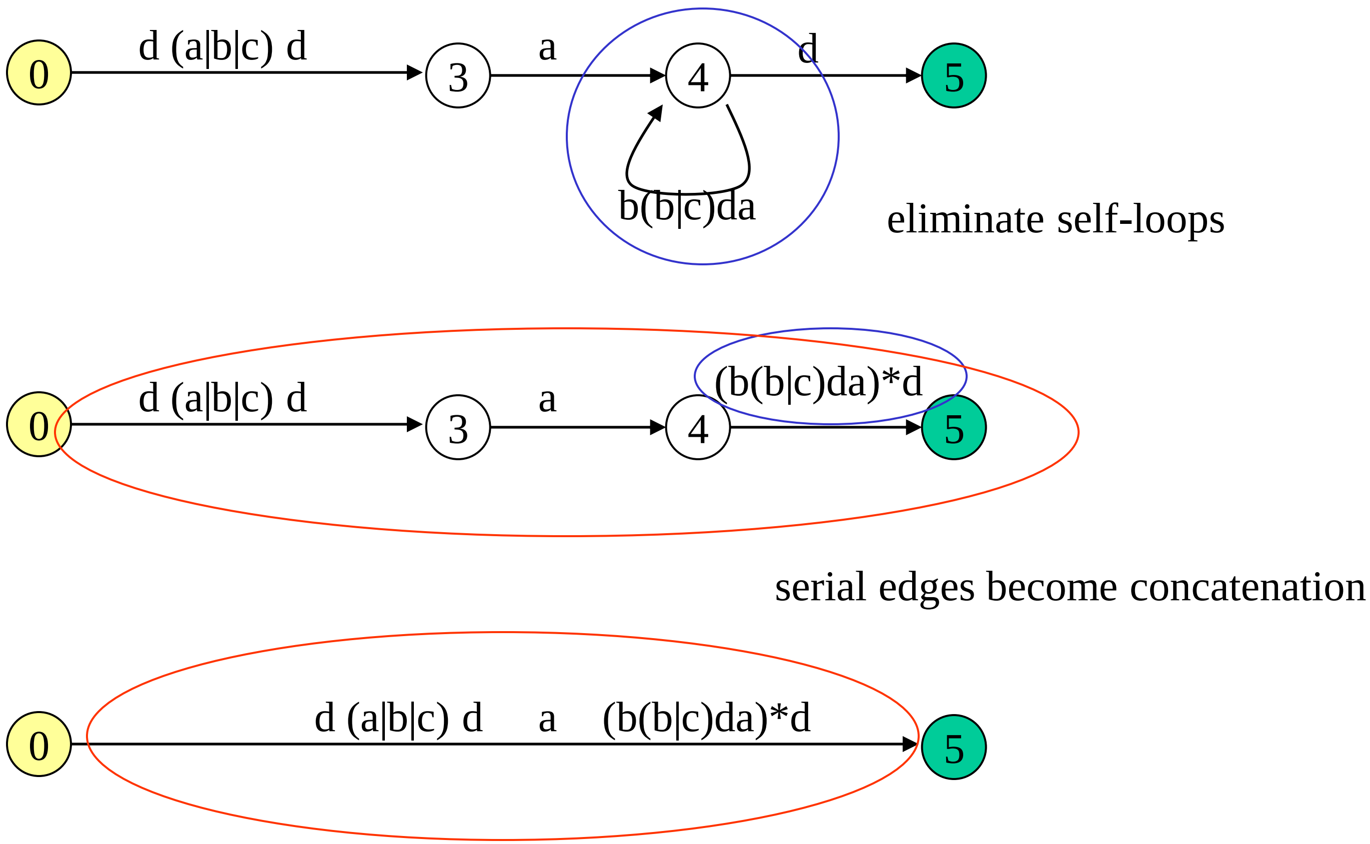
Example



Find paths that can be "shortened"



Example



0
d (a|b|c) d
3
a
4
d
5
b(b|c)da
eliminate self-loops
0
d (a|b|c) d
3
a
4
(b(b|c)da)*d
5
serial edges become concatenation
0
d (a|b|c) d   a   (b(b|c)da)*d
5

Describing Regular Languages

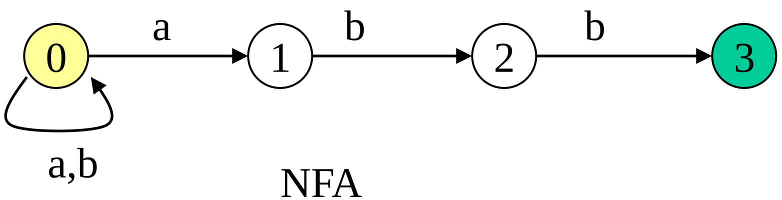
- Generate ***all*** strings in the language
- Generate ***only*** strings in the language

Try the following:

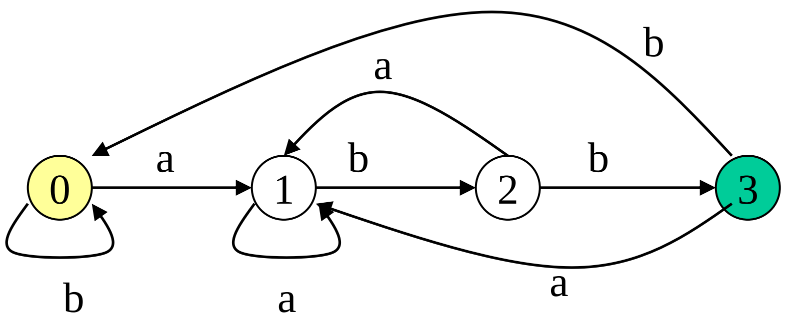
- Strings of $\{a,b\}$ that end with '*abb*'
- Strings of $\{a,b\}$ that don't end with '*abb*'
- Strings of $\{a,b\}$ where every *a* is followed by at least one *b*

Strings of (a|b)* that end in abb

re: (a|b)*abb



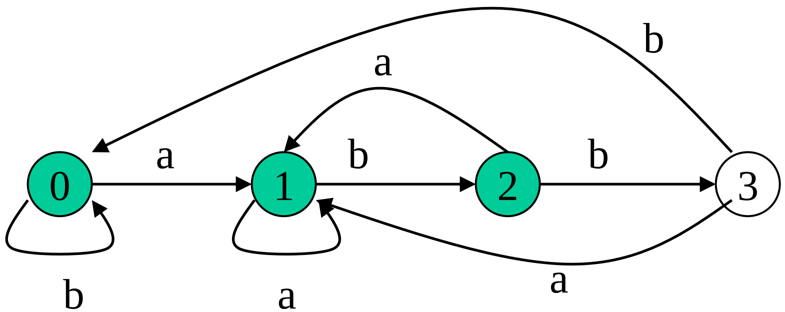
NFA



DFA

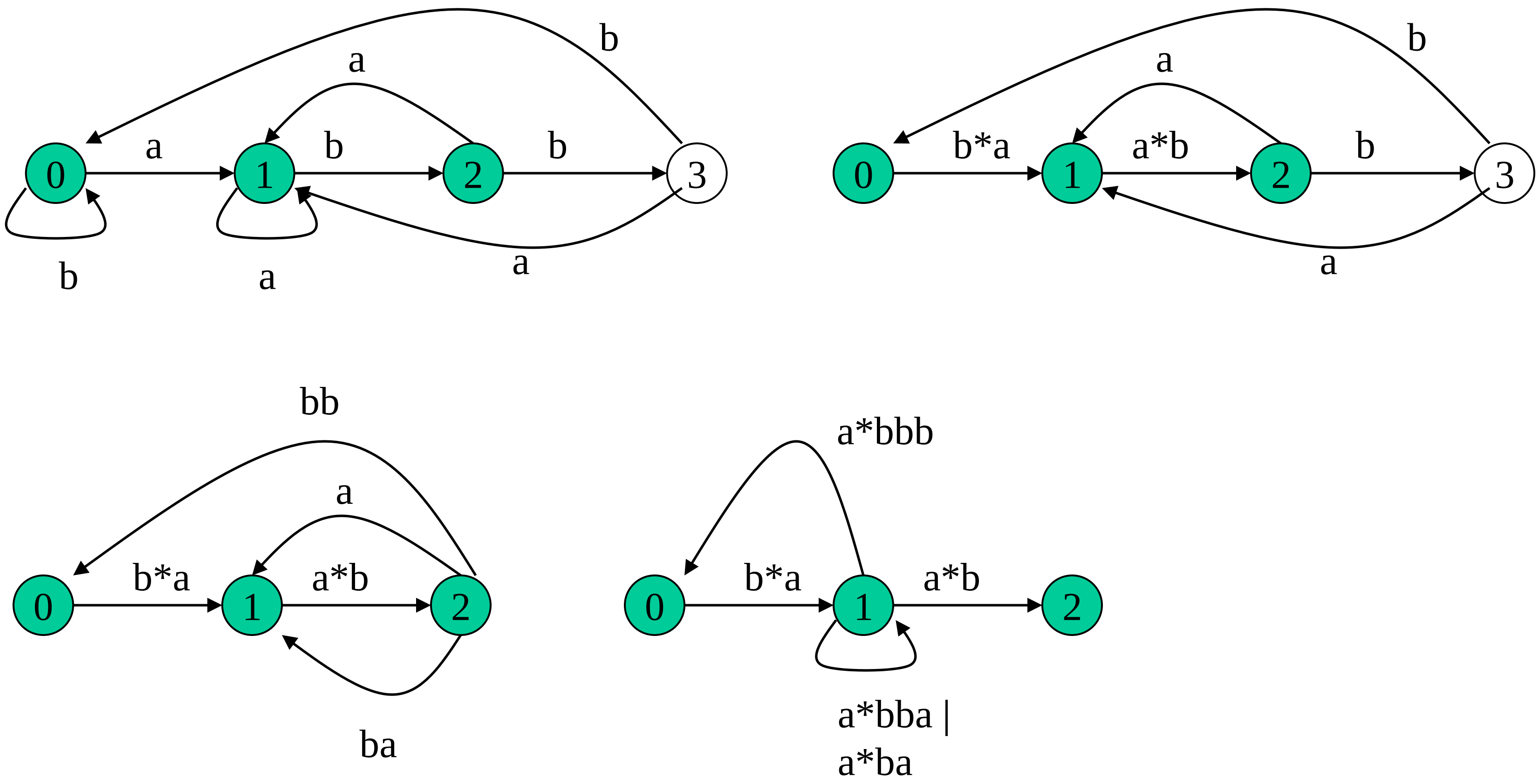
Strings of (a|b)* that don't end in abb

re: ??



DFA/NFA

Strings of (a|b)* that don't end in abb



Suggestions for writing NFA/DFA/RE

- Typically, one of these formalisms is more natural for the problem. Start with that and convert if necessary.
- In NFA/DFAs, each state typically captures some partial solution
- Be sure that you include all relevant edges (ask – does every state have an outgoing transition for all alphabet symbols?)

Non-Regular Languages

Not all languages are regular”

- The language *ww* where *w*=(a|b)*

Non-regular languages cannot be described using REs, NFAs and DFAs.