CT 22001
Compiler Construction

Soma Ghosh - gsn.comp@coeptech.ac.in

Pre-requisite courses

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

The Course covers:

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

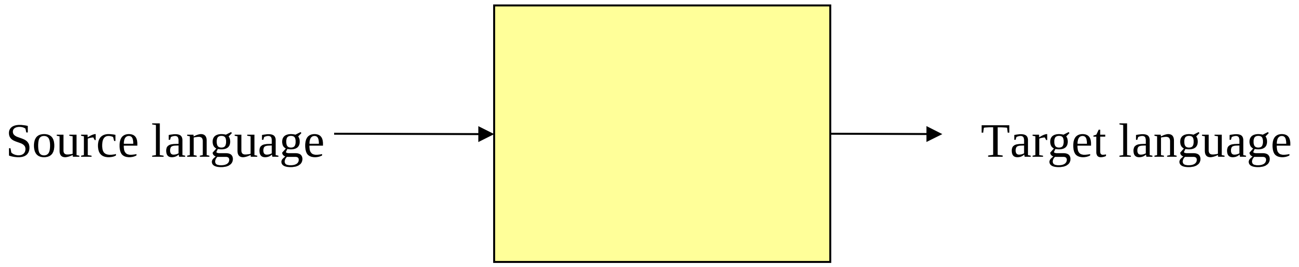
Resources

- 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)

Lecture 1: Introduction to Language Processing & Lexical Analysis

What is a compiler?

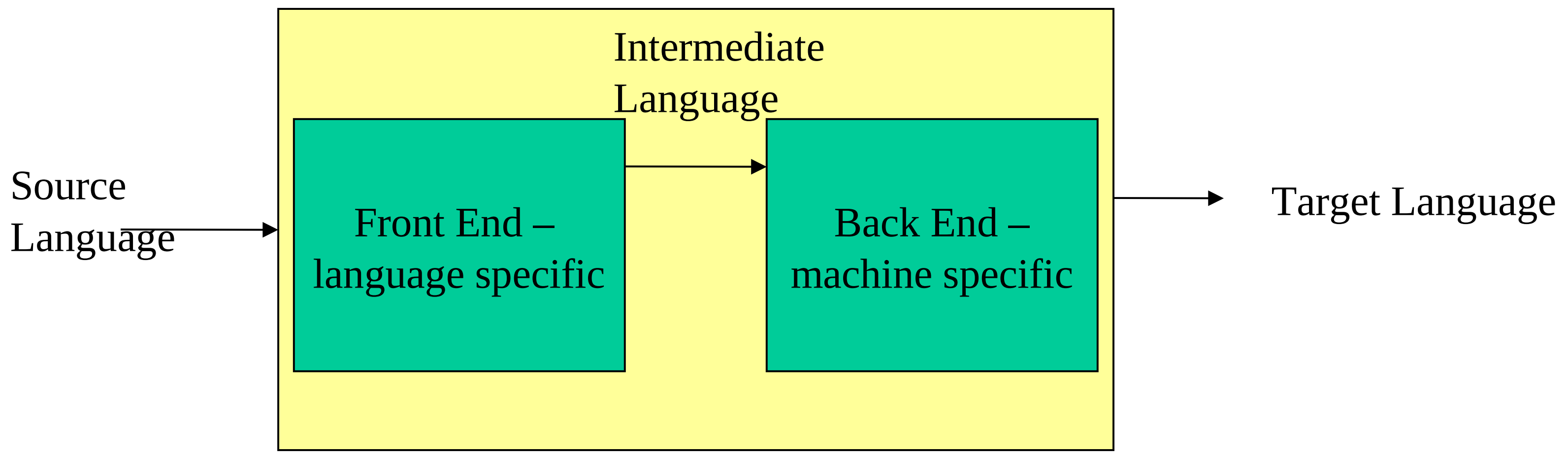
A program that reads a program written in one language and translates it into another language.



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

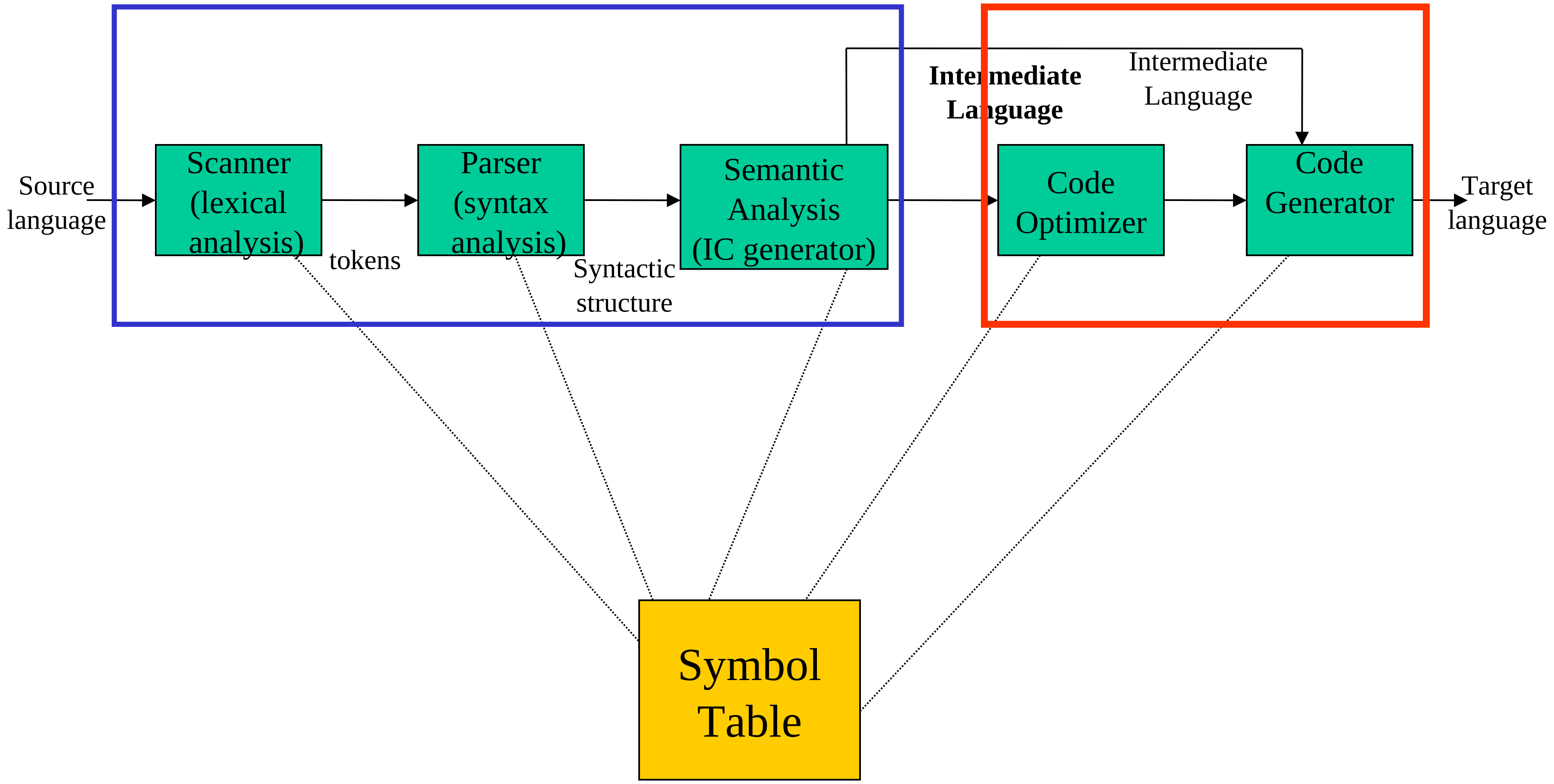
Compiler Architecture

In more detail:

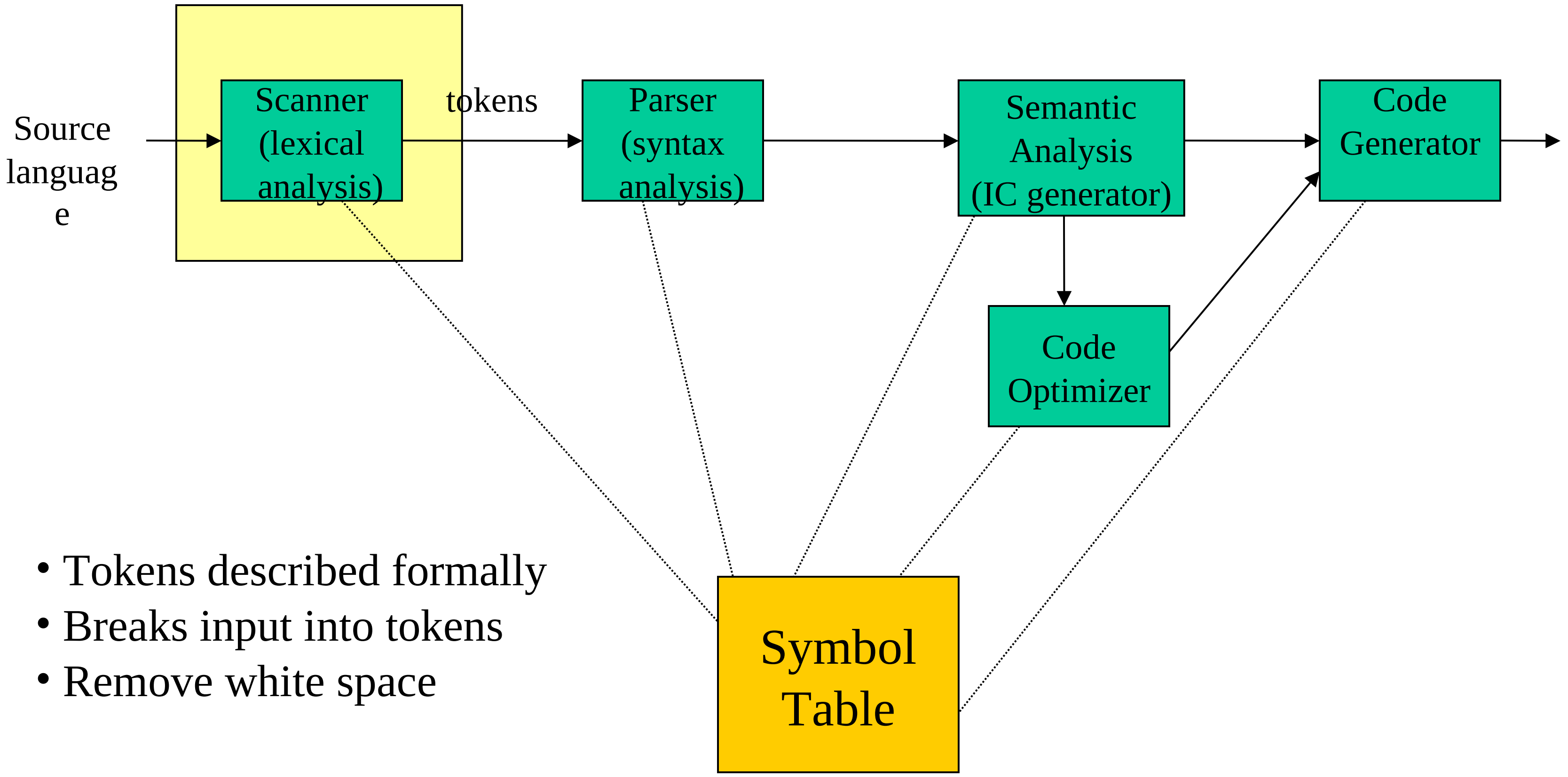


- Separation of Concerns
- Retargeting

Compiler Architecture



Lexical Analysis - Scanning

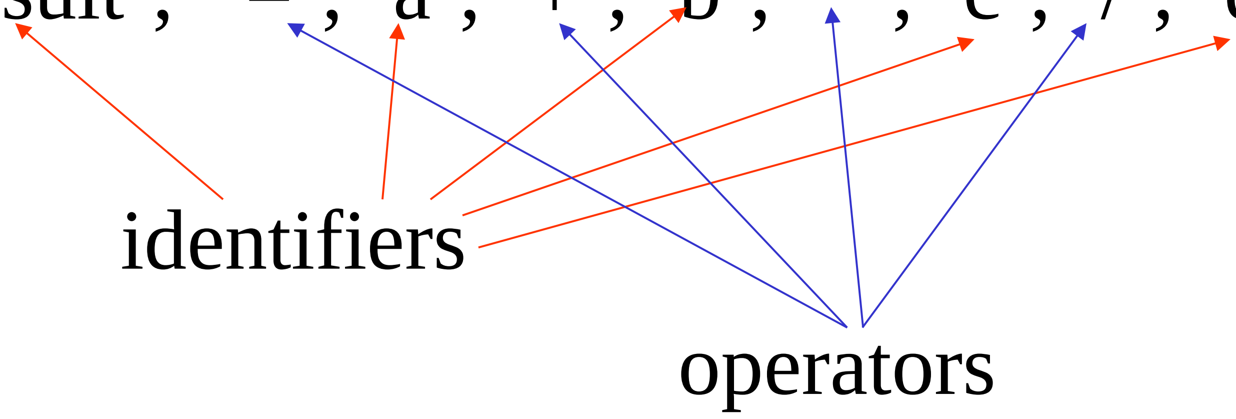


- Tokens described formally
- Breaks input into tokens
- Remove white space

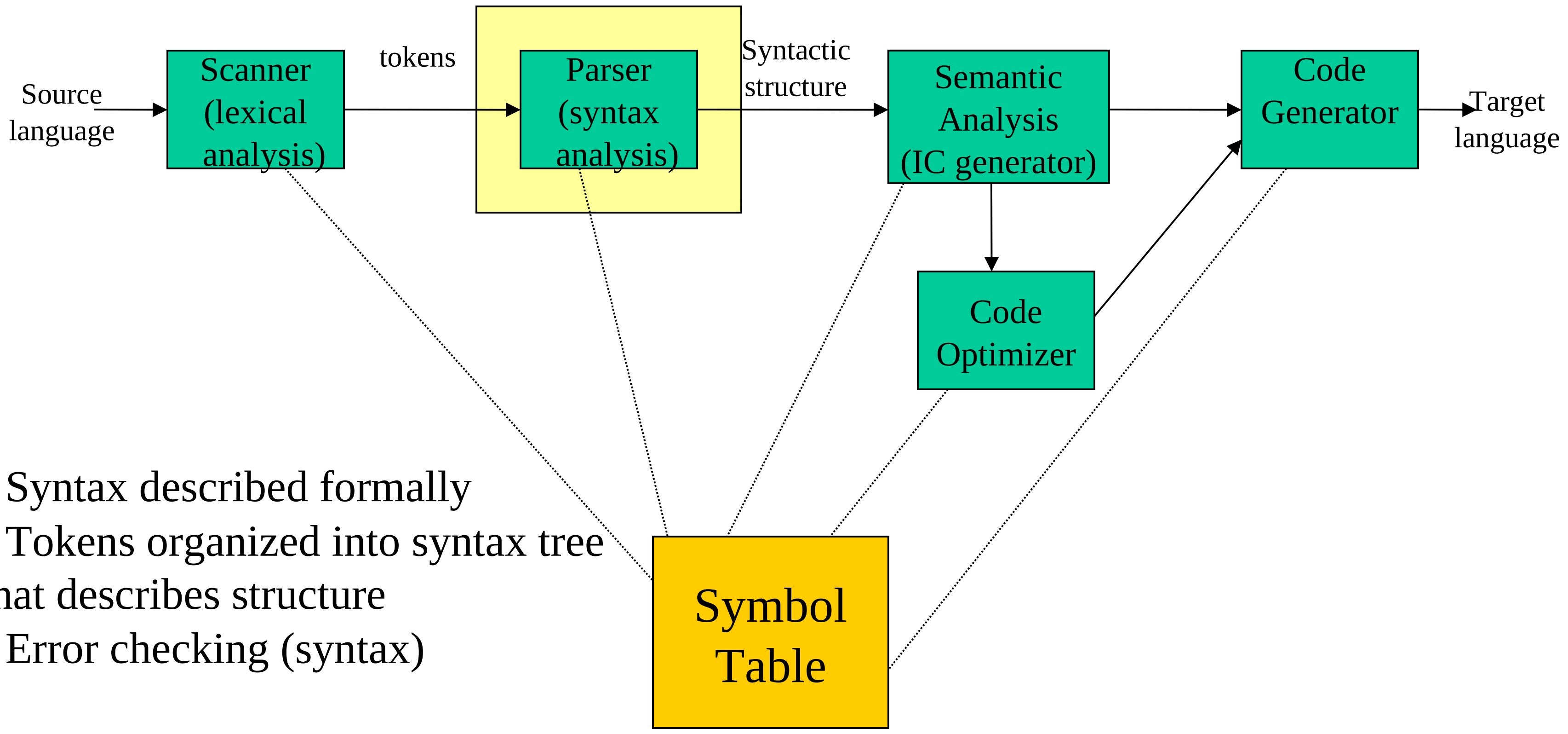
Input: result = a + b * c / d

- Tokens:

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



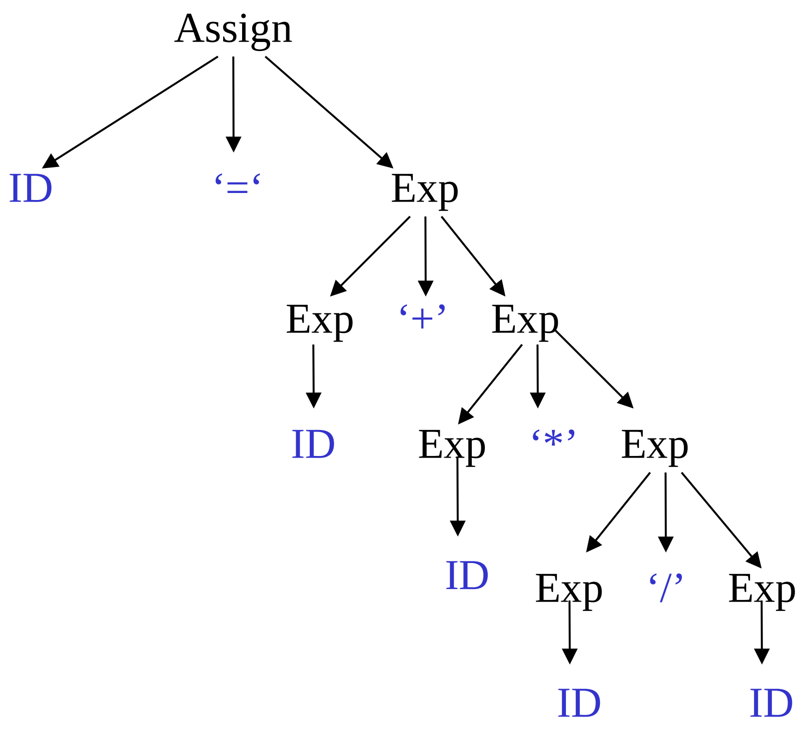
Static Analysis - Parsing



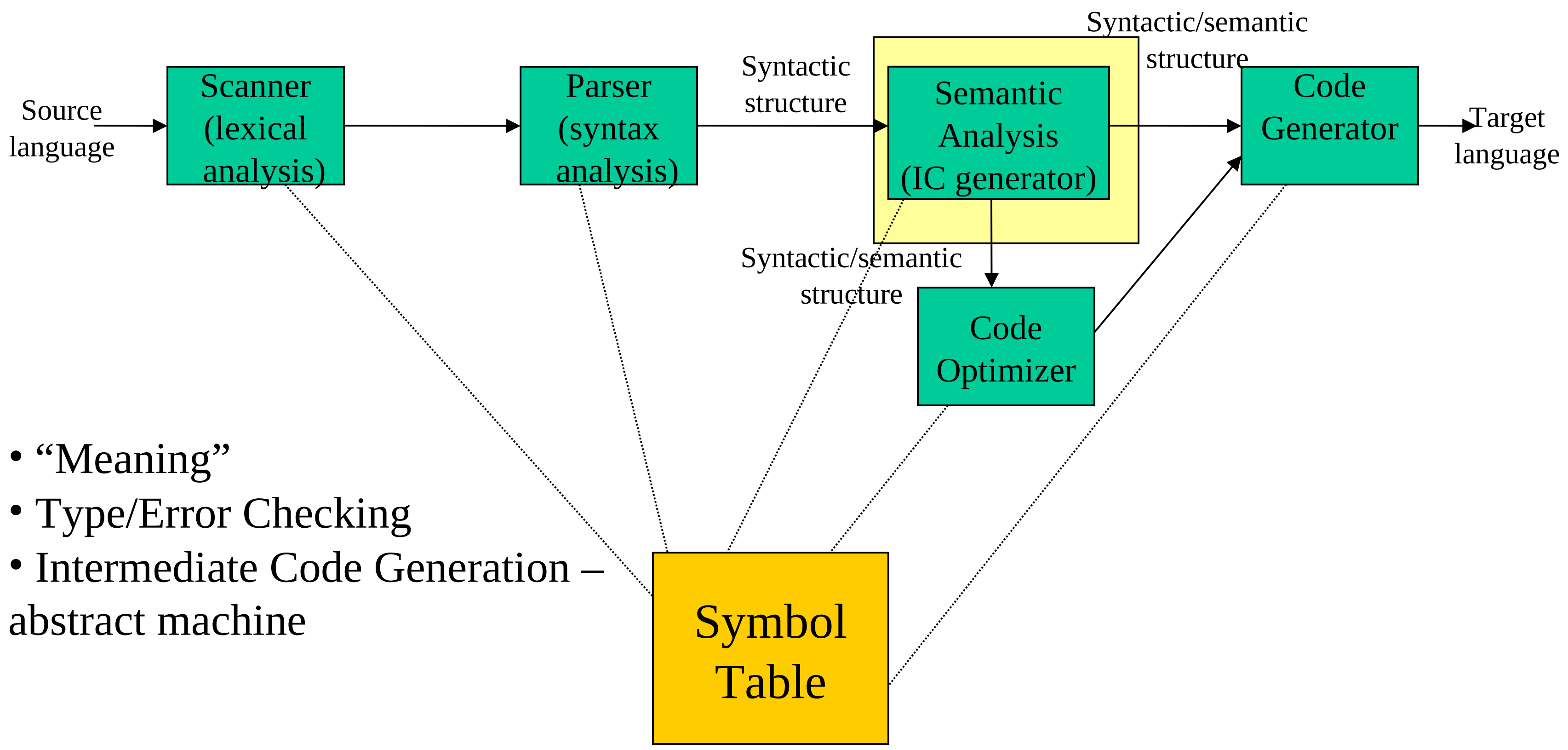
- Syntax described formally
- Tokens organized into syntax tree that describes structure
- Error checking (syntax)

Input: result = a + b * c / d

```
Exp     ::=  Exp ‘+’ Exp
             |    Exp ‘-’ Exp
             |    Exp ‘*’ Exp
             |    Exp ‘/’ Exp
             |    ID
Assign  ::=    ID ‘=‘ Exp
```

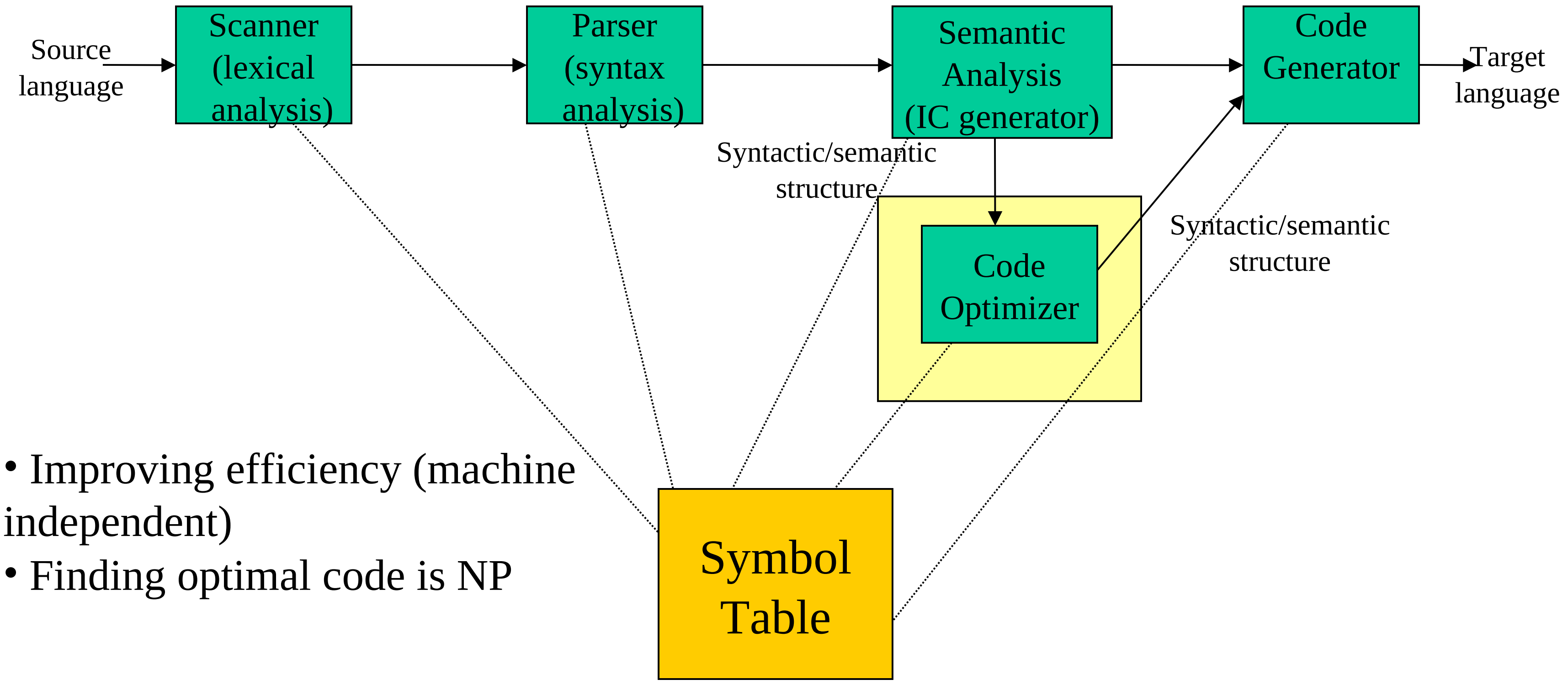


Semantic Analysis


Source language
Scanner (lexical analysis)
Parser (syntax analysis)
Syntactic structure
Semantic Analysis (IC generator)
Syntactic/semantic structure
Code Generator
Target language
Syntactic/semantic structure
Code Optimizer
Symbol Table

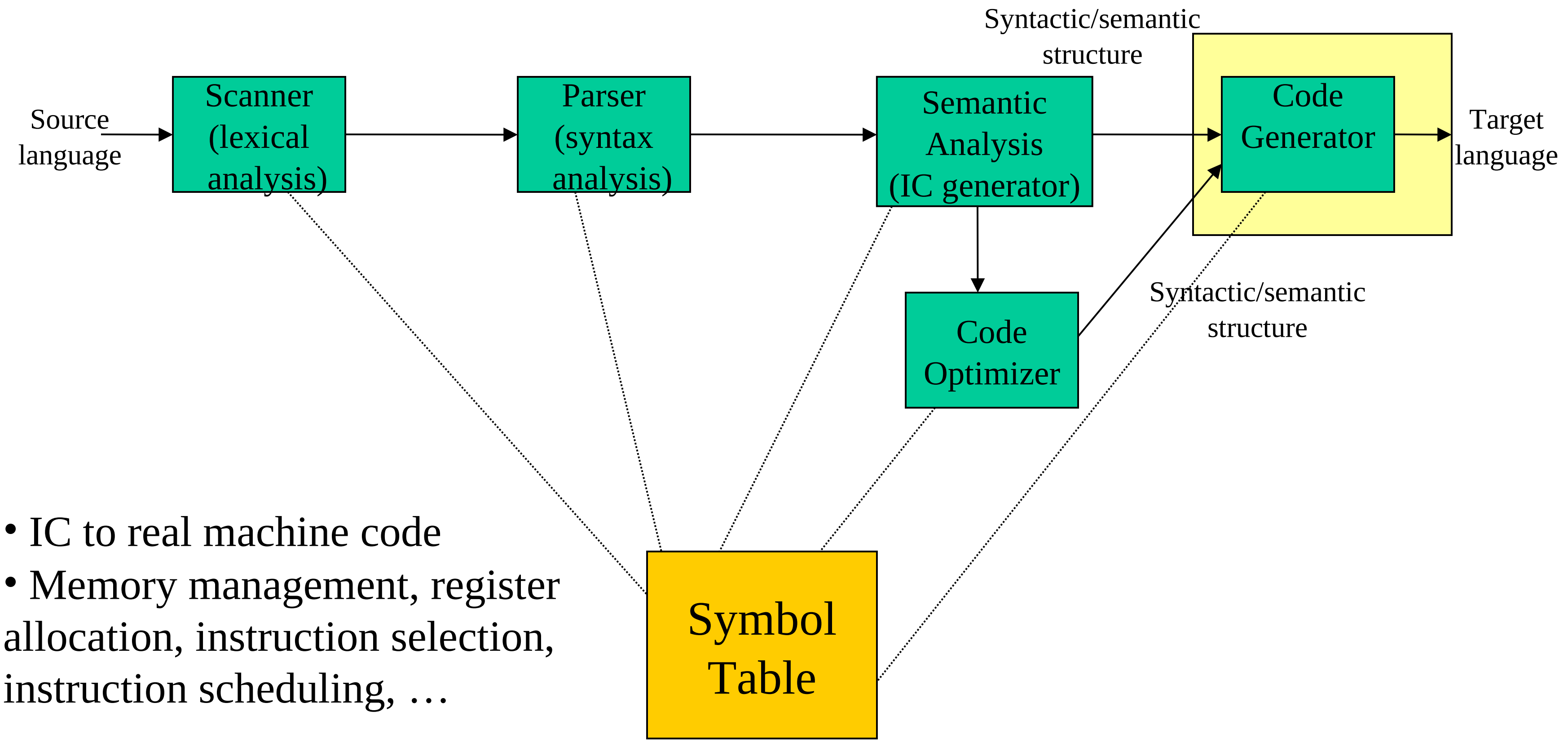
- "Meaning"
- Type/Error Checking
- Intermediate Code Generation – abstract machine

Optimization


Source language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic Analysis (IC generator)
Code Generator
Target language
Syntactic/semantic structure
Code Optimizer
Syntactic/semantic structure
Symbol Table

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

Code Generation


Syntactic/semantic structure
Source language
Scanner (lexical analysis)
Parser (syntax analysis)
Semantic Analysis (IC generator)
Code Generator
Target language
Code Optimizer
Syntactic/semantic structure
Symbol Table

- 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 $\Sigma = \{a,b,c\}$. 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:

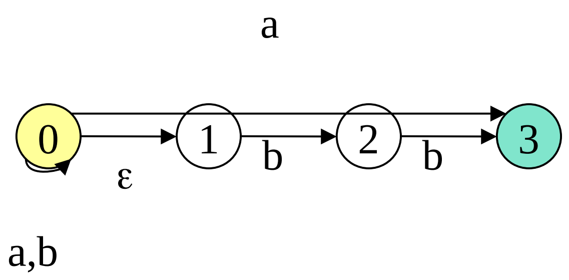
   S x {Σ + ε} → set of S

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



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

$S_0 = 0$

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

$F = \{3\}$

Transition Table:

STATE	a	b	ε
0	0,3	0	1
1		2	
2		3	
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 && $s_0$ is a final state) return Yes;
   if (all input processed && $s_0$ not a final state) return No;

   for all states $s_1$ where transition($s_0$,table[input]) = $s_1$
        if (NFA($s_1$,input_element+1) == Yes) return Yes;

   for all states $s_1$ where transition($s_0$,$\varepsilon$) = $s_1$
        if (NFA($s_1$,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

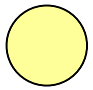
1. There is an algorithm for converting any RE into an NFA.
2. There is an algorithm for converting any NFA to a DFA.
3. There is an algorithm for converting any DFA to a RE.

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

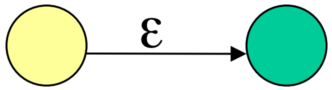
Converting Regular Expressions to NFAs

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

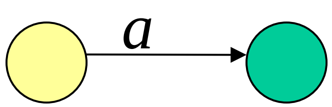
- { } (empty set) is a regular expression for the empty set



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



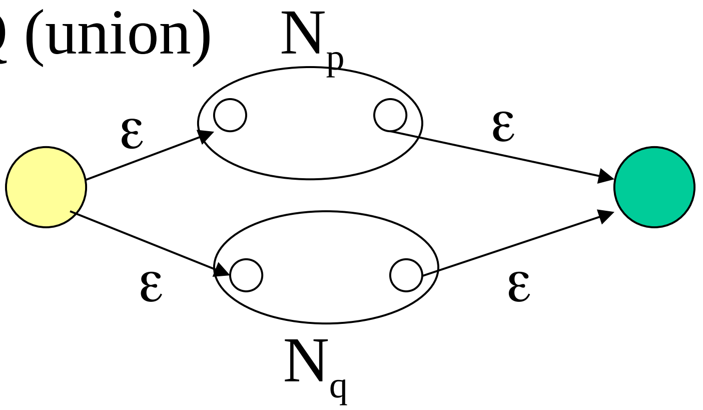
- *a* is a regular expression denoting {*a* } for any *a* in Σ



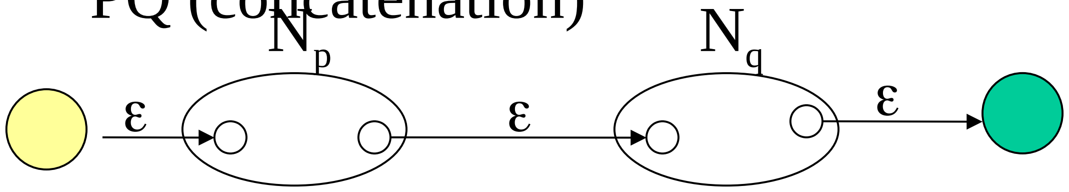
Converting Regular Expressions to NFAs

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

P | Q (union)



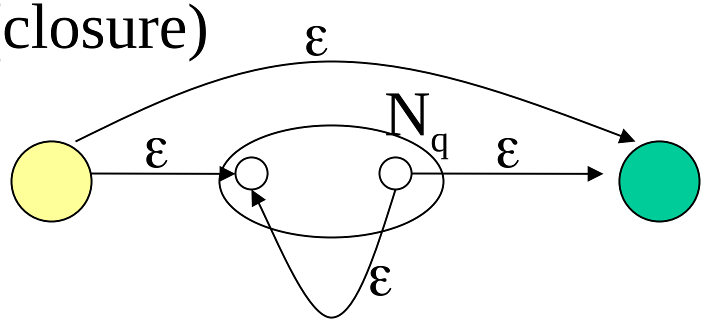
PQ (concatenation)



Converting Regular Expressions to NFAs

If Q is a regular expression with NFA N_q:

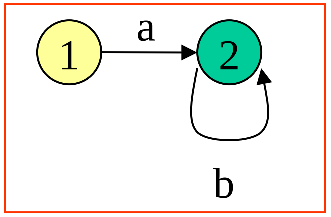
Q* (closure)



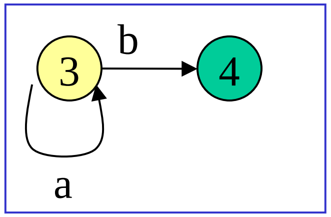
Example (ab* | a*b)*

Starting with:

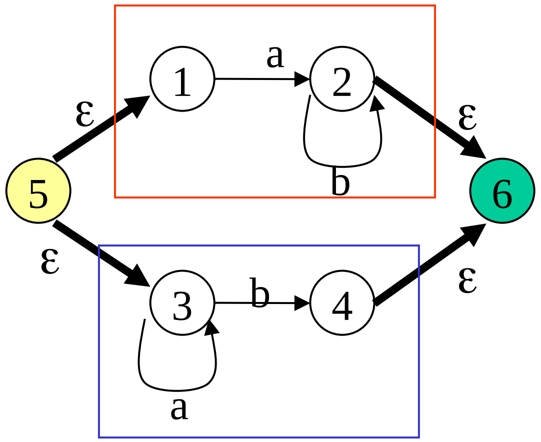
ab*



a*b

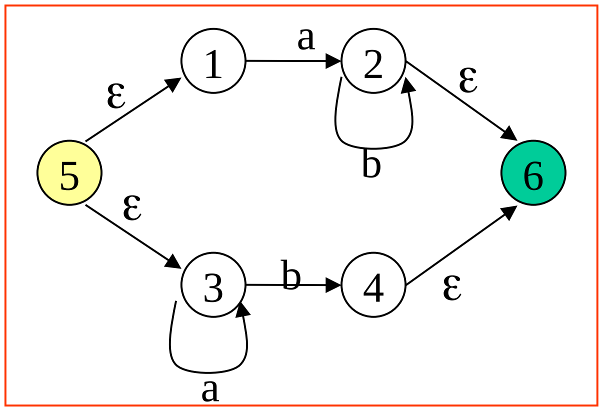


ab* | a*b

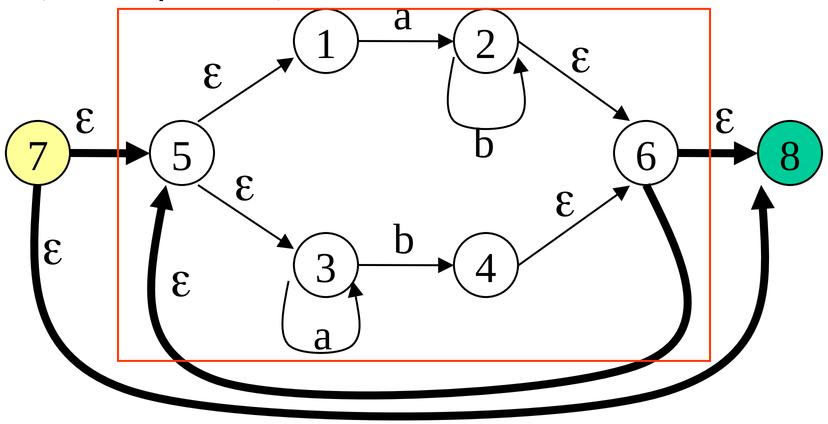


Example (ab* | a*b)*

ab* | a*b



(ab* | a*b)*



Converting NFAs to DFAs

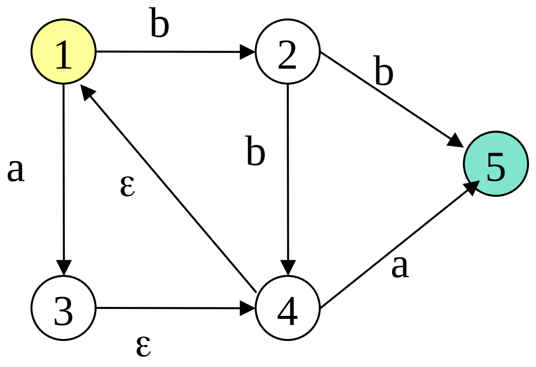
- **Idea**: Each state in the new DFA will correspond to some set of states from the NFA. The DFA will be in state $\{s_0,s_1,\ldots\}$ after input if the NFA could be in *any* of these states for the same input.

- **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

$s_D = \varepsilon$-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.



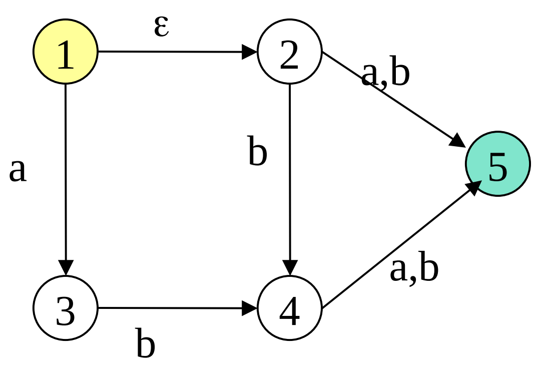
ε-closure({1,2,5}) = {1,2,5}
ε-closure({4}) = {1,4}
ε-closure({3}) = {1,3,4}
ε-closure({3,5}) = {1,3,4,5}

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 $F_N$
```

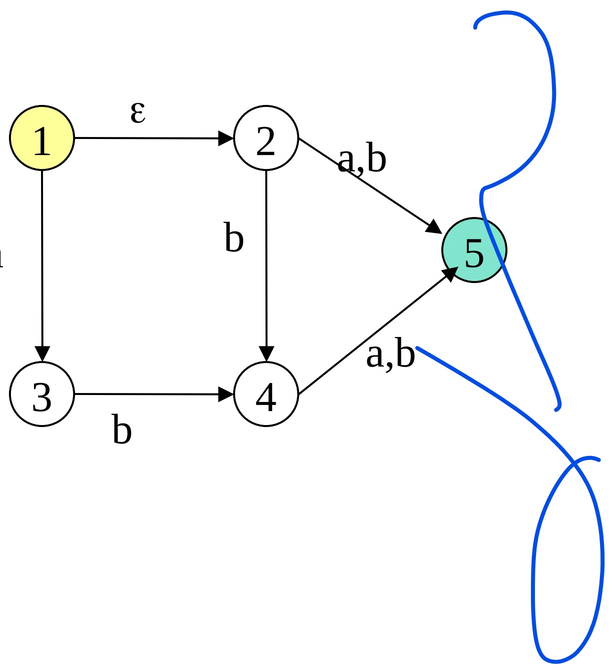
Example 1: Subset Construction

NFA


$$\begin{tikzcd}
1 \arrow[r, "\varepsilon"] \arrow[d, "a"'] & 2 \arrow[d, "b"'] \arrow[r, "{a,b}"] & 5 \\
3 \arrow[r, "b"'] & 4 \arrow[ur, "{a,b}"'] &
\end{tikzcd}$$

Example 1: Subset Construction

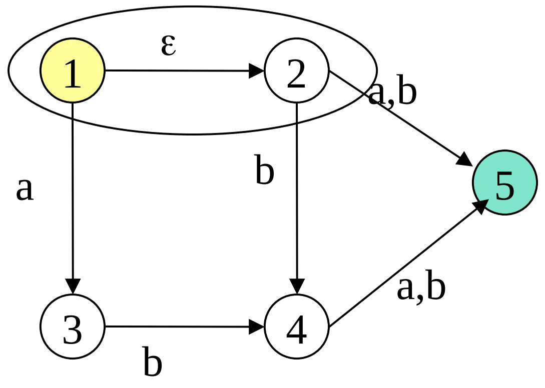
ε-NFA to DFA

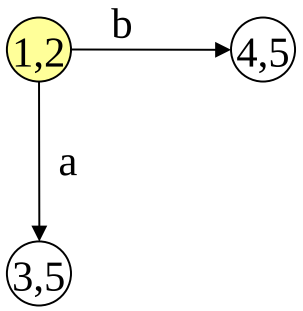


	a	b
{1,2}		

Example 1: Subset Construction

NFA

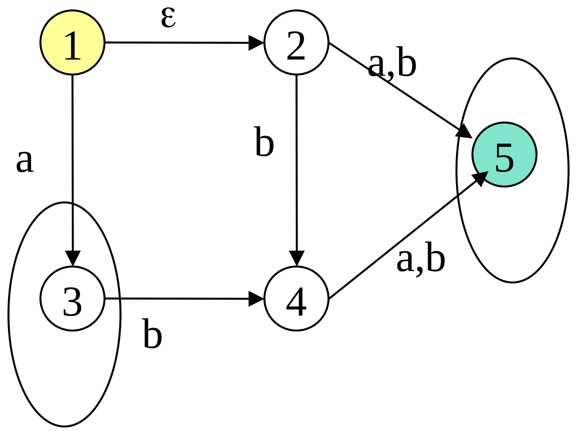


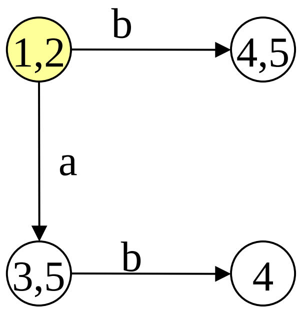


	a	b
{1,2}	{3,5}	{4,5}
{3,5}		
{4,5}		

Example 1: Subset Construction

NFA

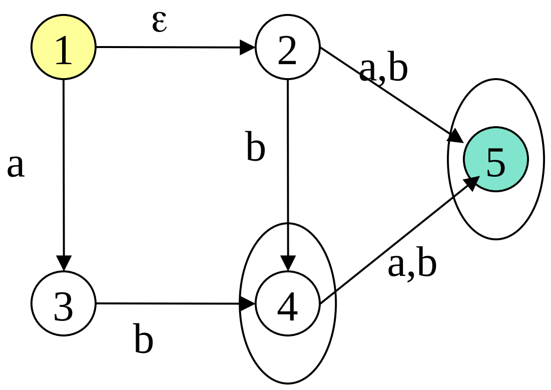


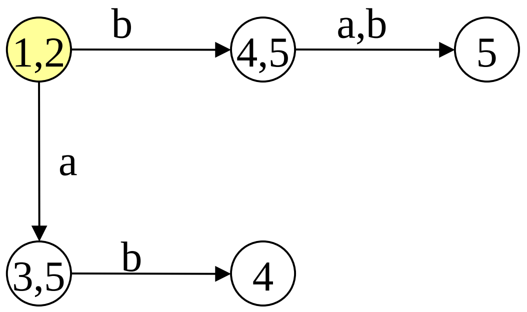


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}		
{4}		

Example 1: Subset Construction

NFA

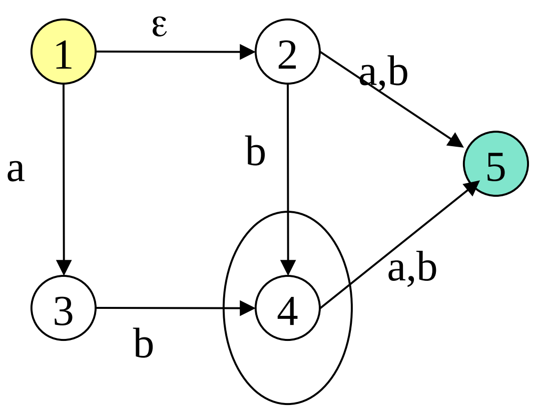


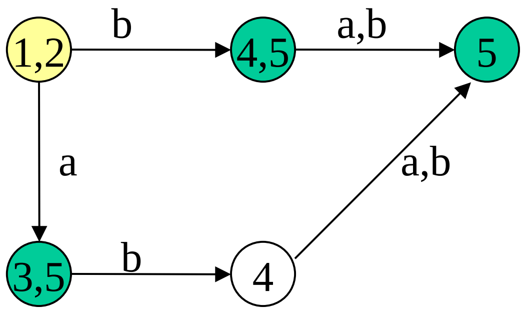


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}		
{5}		

Example 1: Subset Construction

NFA



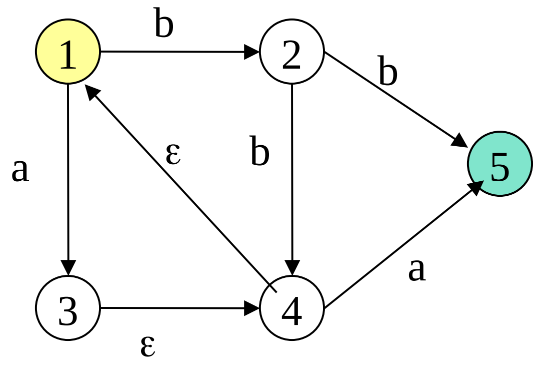


	a	b
{1,2}	{3,5}	{4,5}
{3,5}	-	{4}
{4,5}	{5}	{5}
{4}	{5}	{5}
{5}	-	-

All final states since the NFA final state is included

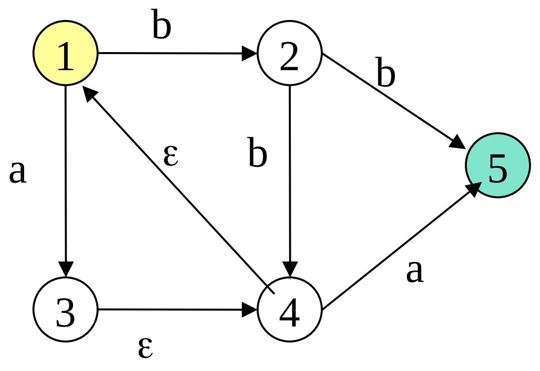
Example 2: Subset Construction

NFA

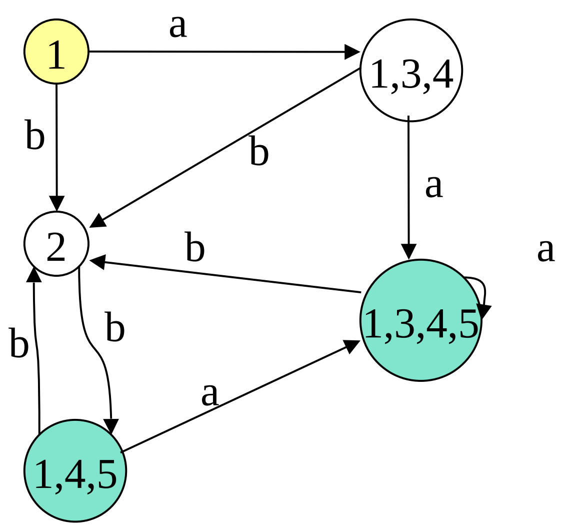

$$\begin{tikzcd}
1 \arrow[r, "b"] \arrow[d, "a"'] & 2 \arrow[rd, "b"] \arrow[d, "b"'] & \\
3 \arrow[r, "\varepsilon"'] & 4 \arrow[lu, "\varepsilon"] \arrow[r, "a"'] & 5
\end{tikzcd}$$

Example 2: Subset Construction

NFA

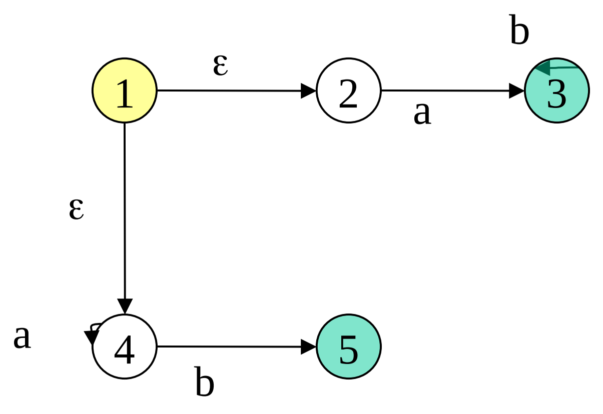


DFA

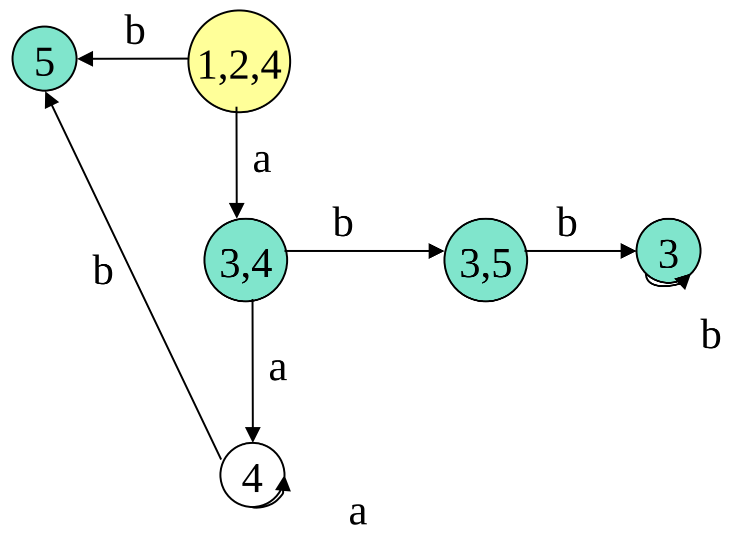


Example 3: Subset Construction

NFA



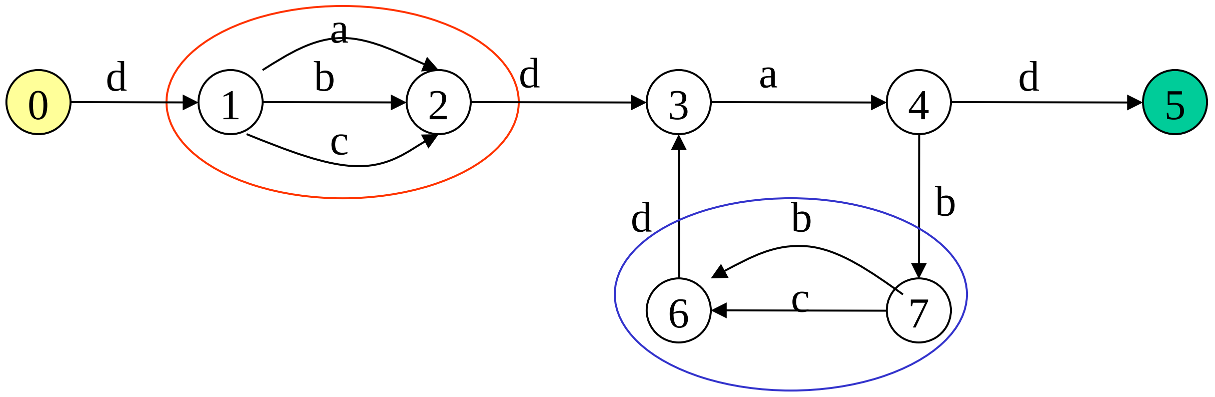
DFA



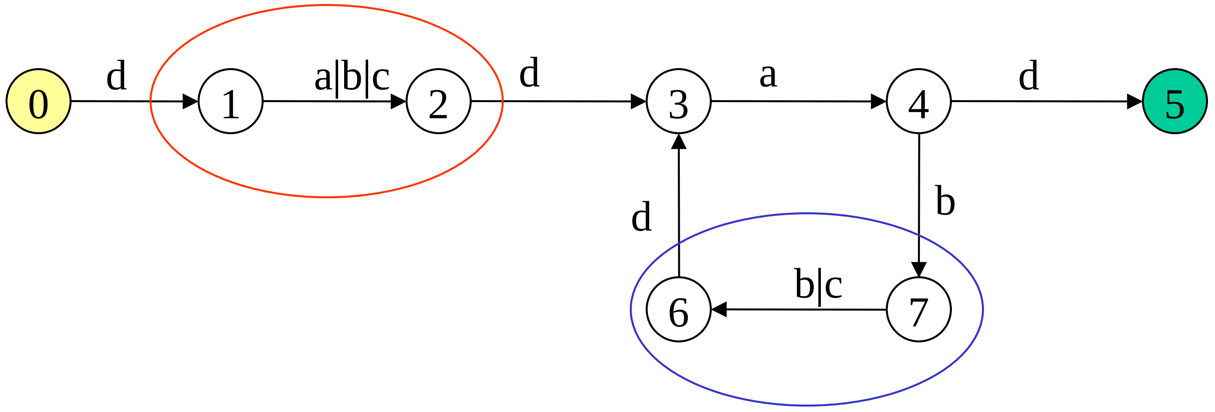
Converting DFAs to REs

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.

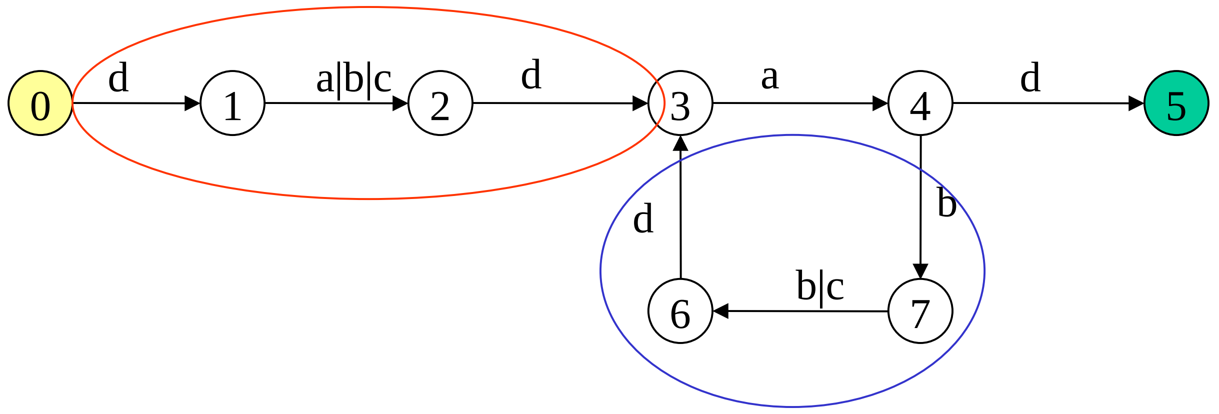
Example



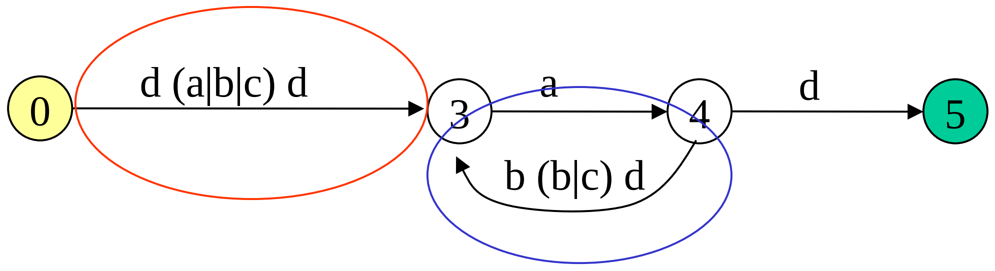
parallel edges become alternation



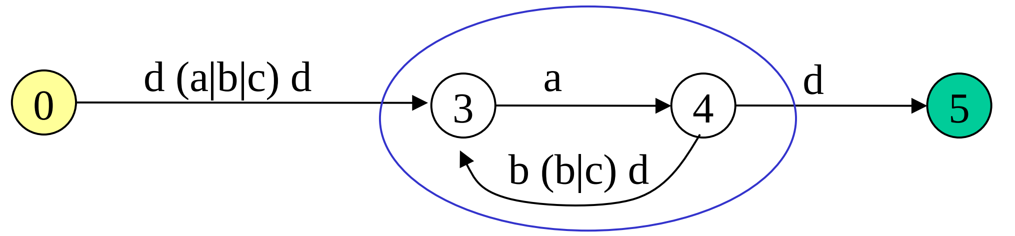
Example



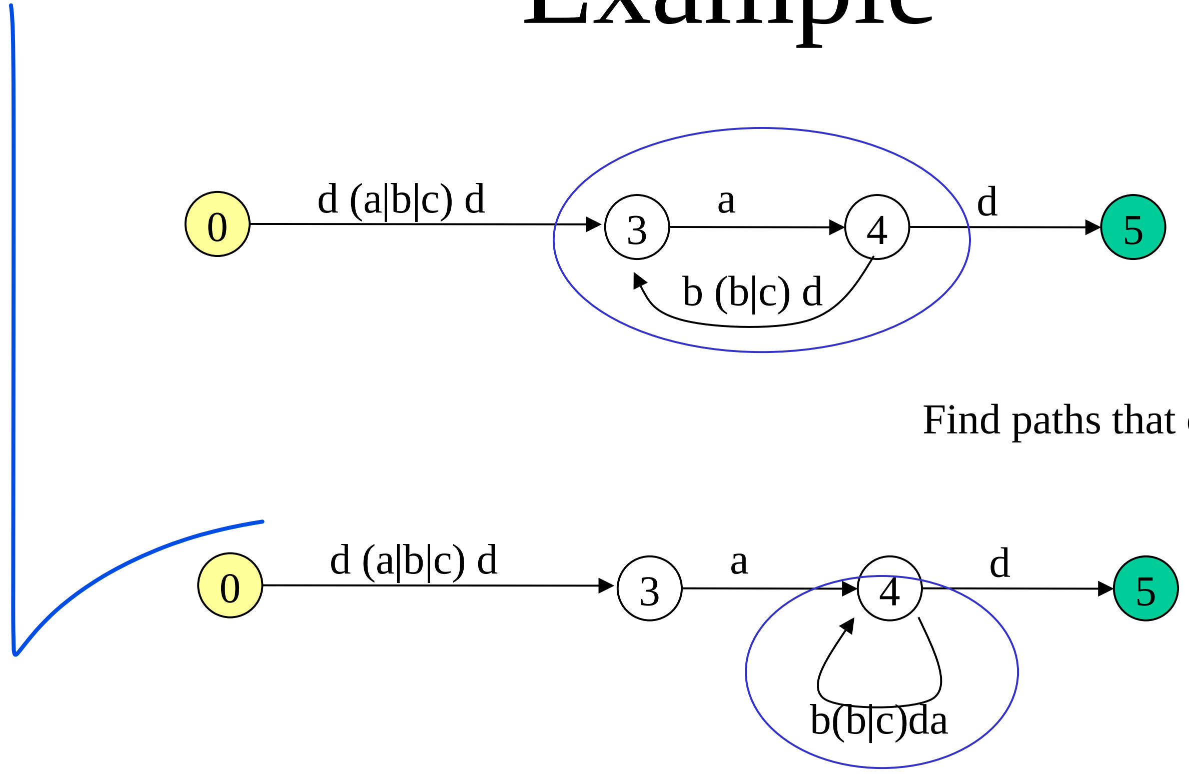
serial edges become concatenation



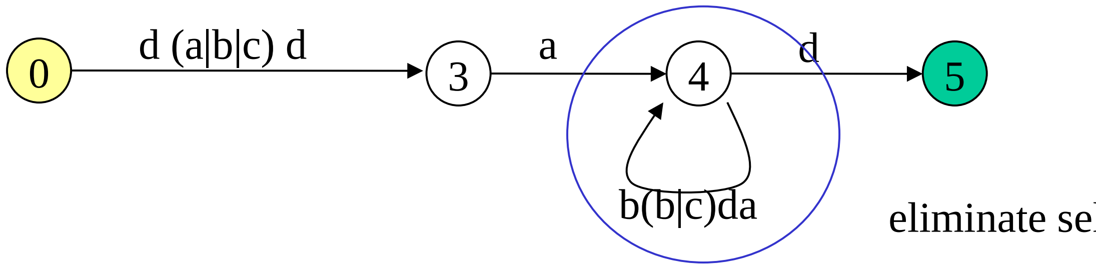
Example



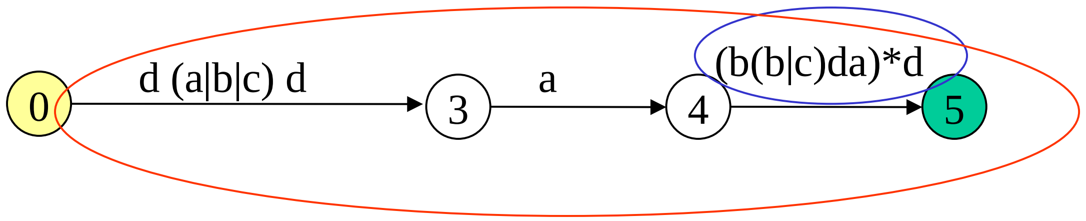
Find paths that can be "shortened"



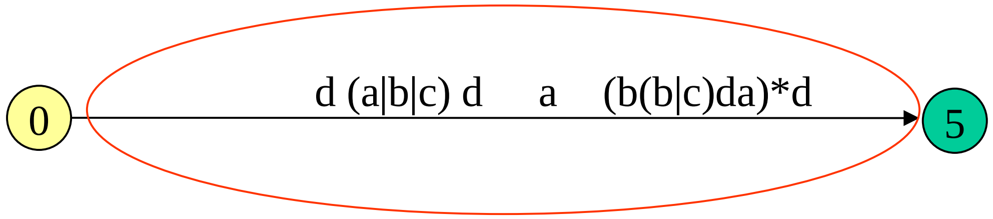
Example



eliminate self-loops



serial edges become concatenation



Describing Regular Languages

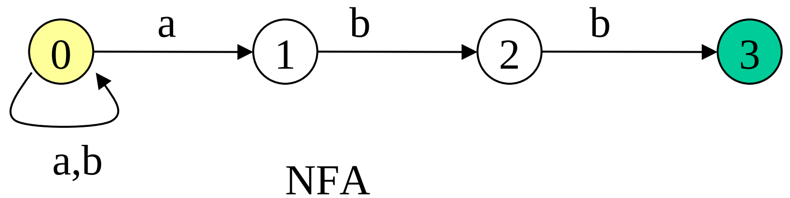
- Generate ***all*** strings in the language
- Generate ***only*** strings in the language

Try the following:

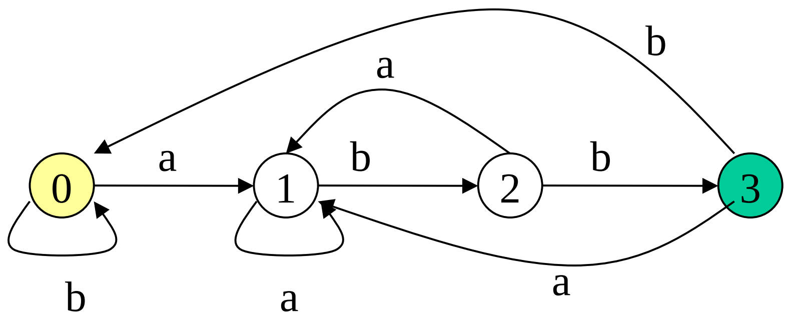
- Strings of {*a*,*b*} that end with '*abb*'
- Strings of {*a*,*b*} that don't end with '*abb*'
- Strings of {*a*,*b*} where every *a* is followed by at least one *b*

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

re: (a|b)*abb



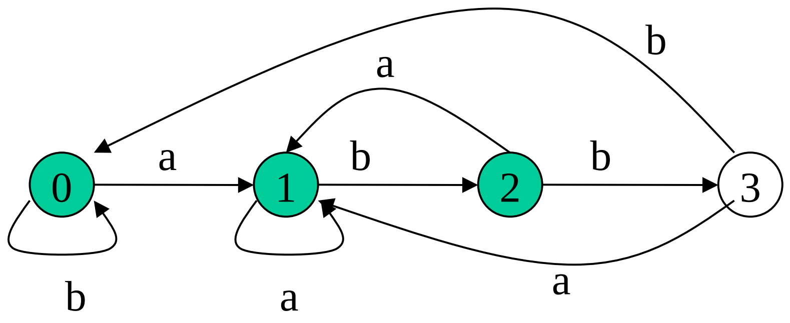
NFA



DFA

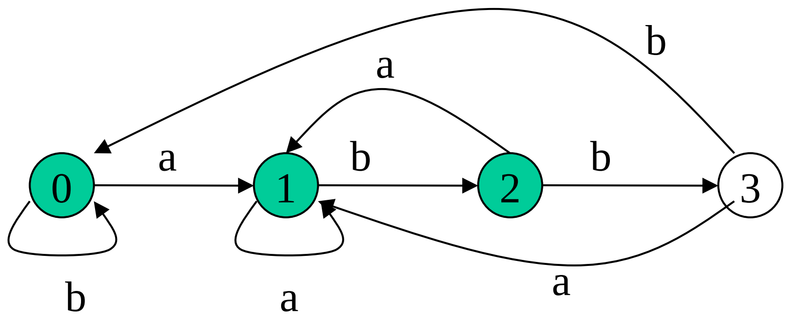
Strings of (a|b)* that don't end in abb

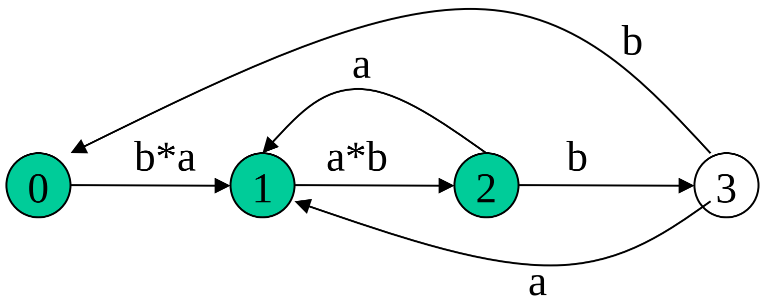
re: ??

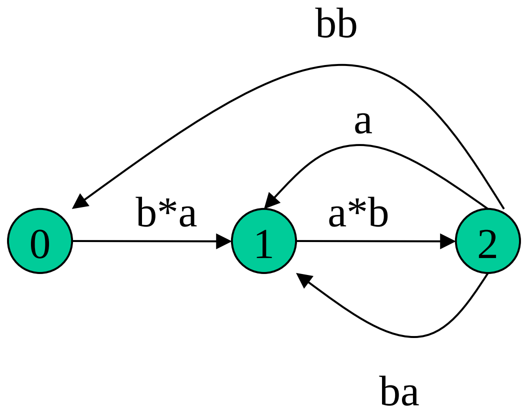


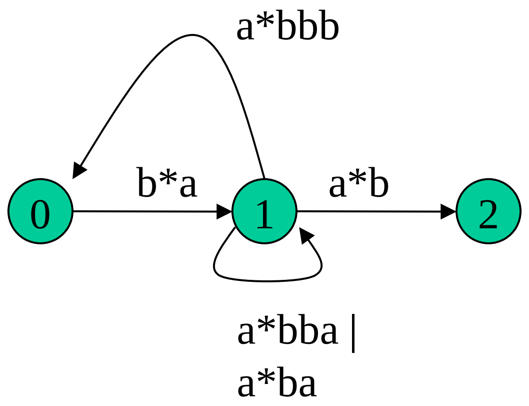
DFA/NFA

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)*

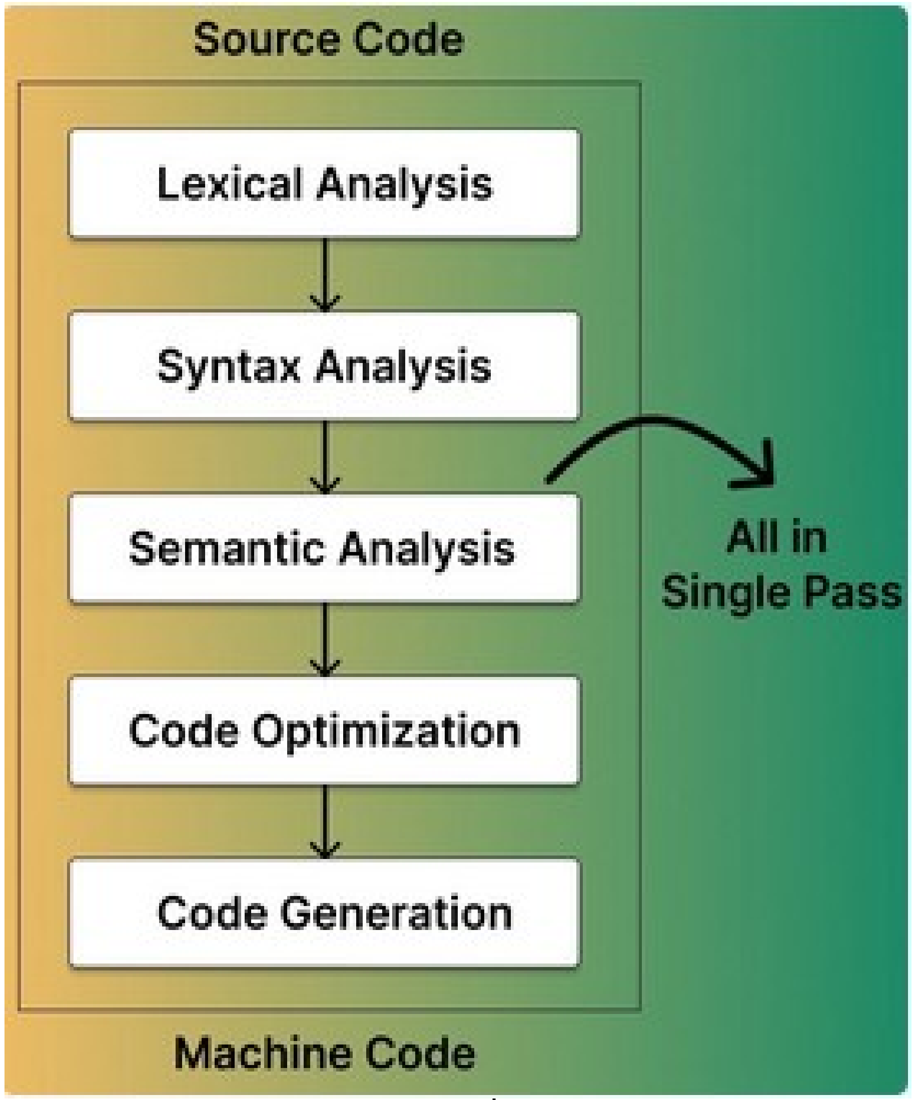
Non-regular languages cannot be described using REs, NFAs and DFAs.

Single Pass and Multi Pass Compiler

The compiler converts source code into machine code

single pass and multi pass compilers are two types of compilers.

Single Pass Compiler



Single Pass Compiler

- Firstly, the compiler scans the source code during lexical analysis and divides it into similar tokens.
- Then, with the help of the Parse tree, it performs syntax analysis, verifying the code with the programming language's grammar.
- Finally, it generates the machine code.
- The three main steps are lexical analysis, syntax analysis, and code generation.
- There is no code optimization or intermediate code generation.
- Pascal, C, FORTRAN, etc.

Pros & Cons of Single Pass Compilers

Advantages

- It is faster for small programs and embedded systems in case of small code size and critical time compilation.
- It requires less memory.
- It can generate the machine code faster than another type.

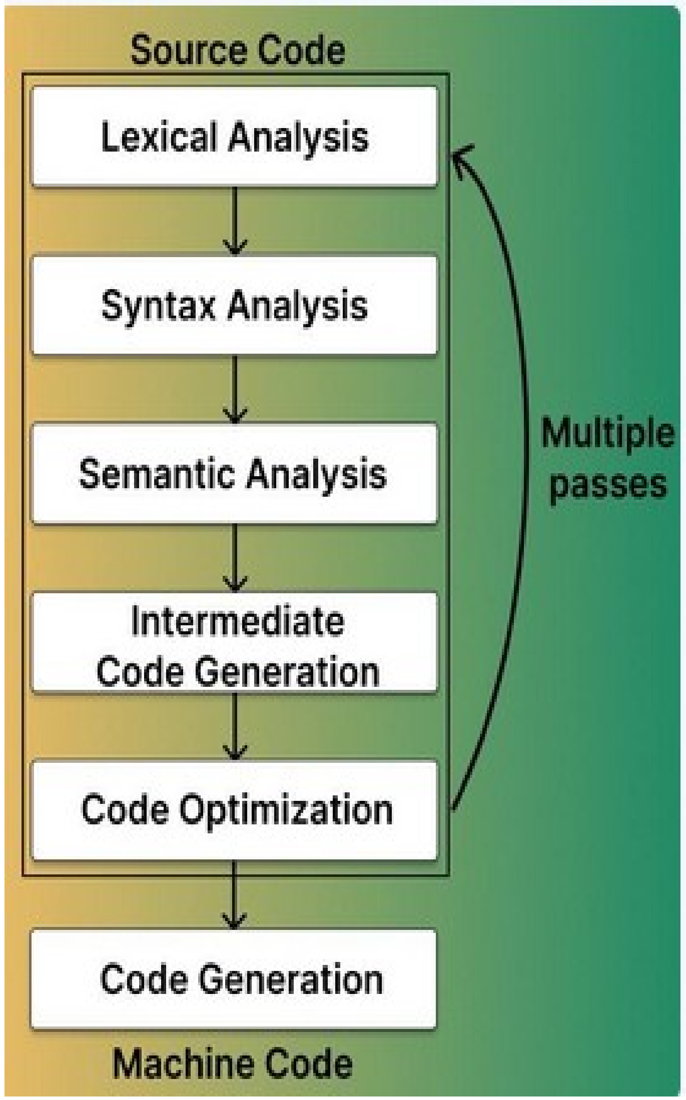
Disadvantages

- It cannot perform complex optimization techniques as we need to perform multiple passes for optimization.
- There is no intermediate code generation.
- It imposes some restrictions upon the program constants, variables, types, and procedures that must be defined before use.

Multi Pass Compiler

- A multi-pass compiler makes the source code go through several passes during the compilation process
- it also generates intermediate optimized code after each step.
- It converts the source program into one or more intermediate codes in steps between the source code and machine code.
- It reprocesses the entire program in each succeeding pass.

Multi Pass Compiler



Multi Pass Compiler

- Each pass takes the result of the previous pass as the input and creates an intermediate optimized output.
- Likewise, the code improves in each pass until the final pass generates the required code.
- A multi-pass compiler performs additional tasks such as intermediate code generation, machine-dependent code optimization, and machine-independent code optimization.
- Some examples of programming languages related to multi-pass compilers are Java, C++, Lisp, GCC etc.
- These languages' codes are highly abstract and require multiple passes to optimize.

Pros & Cons of Multi Pass Compiler

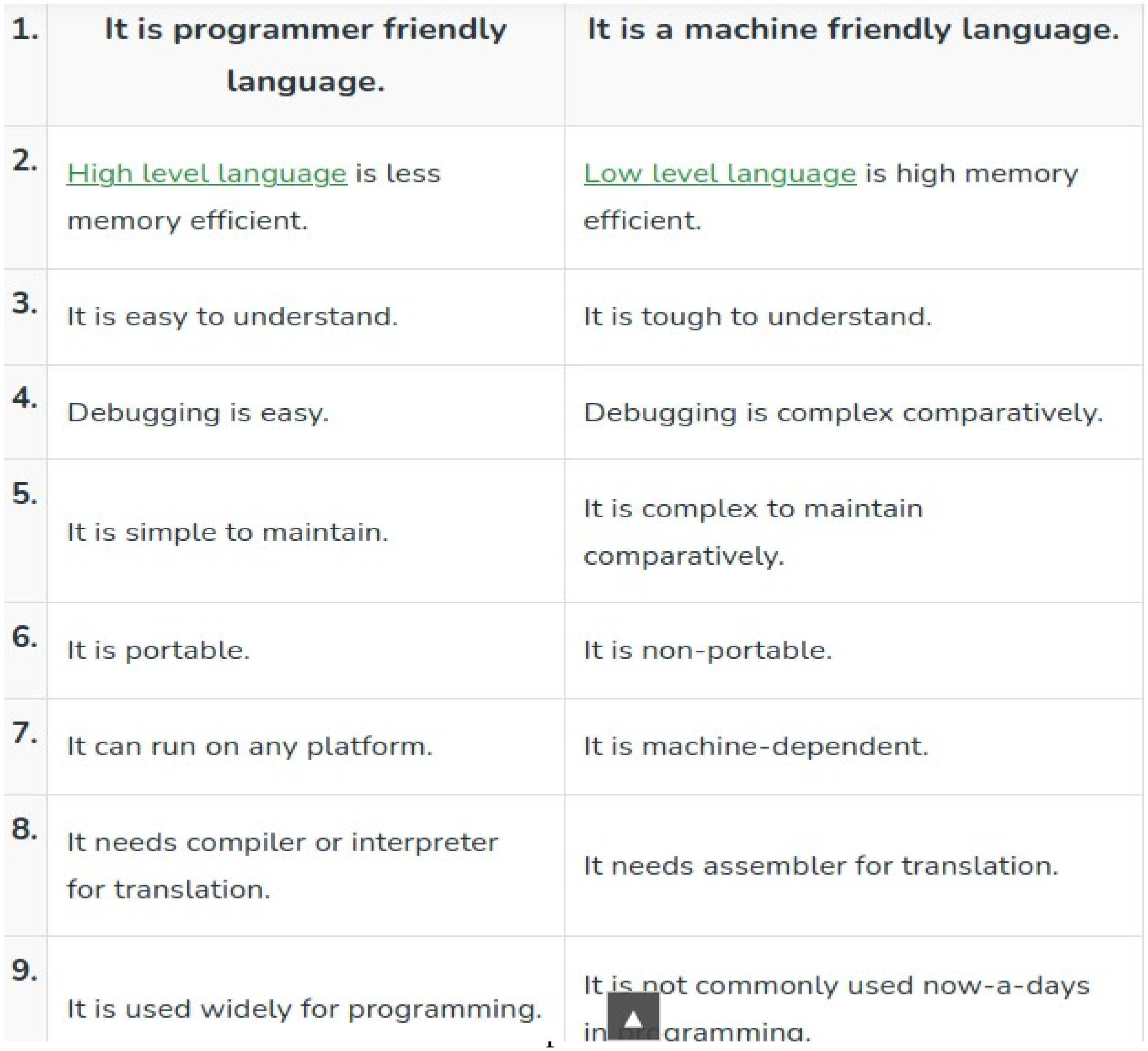
Advantages

- It is machine independent as multiple passes include a modular structure, and code generation is separate from other steps; the passes can be reused for different machines.
- It works well with complex programming languages.
- It generates better code than single-pass compilers, as they perform more optimization through multiple passes.

Disadvantages

- It requires more memory and processing power as it goes through multiple passes over the source code and stores intermediate optimized codes.
- It also increases the compilation time.

Difference between High level & Low level language



1.	**It is programmer friendly language.**	**It is a machine friendly language.**
2.	High level language is less memory efficient.	Low level language is high memory efficient.
3.	It is easy to understand.	It is tough to understand.
4.	Debugging is easy.	Debugging is complex comparatively.
5.	It is simple to maintain.	It is complex to maintain comparatively.
6.	It is portable.	It is non-portable.
7.	It can run on any platform.	It is machine-dependent.
8.	It needs compiler or interpreter for translation.	It needs assembler for translation.
9.	It is used widely for programming.	It is not commonly used now-a-days in programming.