COMPILER DESIGN
CS6109 MODULE 1 & 2

BE COMPUTER SCIENCE AND ENGINEERING

REGULATION 2018 RUSA

Thanasekhar B

COURSE OBJECTIVES : CS6109 COMPILER DESIGN

- To know about the various transformations in the different phases of the compiler, error handling and means of implementing the phases
- To learn about the techniques for tokenization and parsing
- To understand the ways of converting a source language to intermediate representation
- To have an idea about the different ways of generating assembly code
- To have a brief understanding about the various code optimization techniques

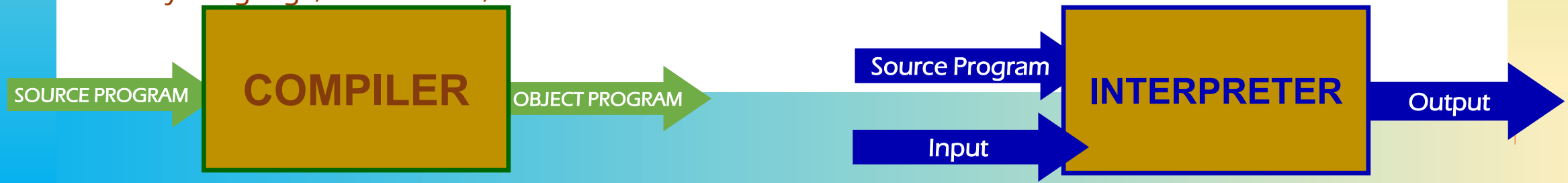
CS6109 COMPILER DESIGN

- MODULE 1
  - Phases of the compiler
  - Compiler construction tools
  - Role of Assemblers, Macro processors,
  - Loaders, Linkers
- MODULE II
  - Role of a Lexical analyser
    - Recognition of Tokens
    - Specification of Tokens
  - Finite Automata (FA)
    - Deterministic Finite Automata (DFA)
    - Non-deterministic Finite Automata (NFA)
    - Finite Automata with Epsilon Transitions
    - NFA to DFA conversion
    - Minimization of Automata.
- Extra Learning Components
  - Programming Language Constructs
  - LEX Programs for Tokenization
  - NFA → DFA
  - $\in$-NFA to DFA

Compiler vs Interpreter

- Compiler is a program that can read a program in one language and translate it into an equivalent program in another language
- Compilers produce executable code in the specified media thus avoiding the compilation for every execution
- Compilers could recover from minor errors thus reporting errors in the whole program immediately
- A compiled program requires very less time for its execution
- Program execution begins only after successful compilation of the whole program
- Object code produced may be another HLL, Assembly Language, absolute ML, relocatable ML

- Interpreter is a translator program that transforms the source program into a simplified language directly for execution
- The intermediate code produced by the interpreters can't be reused, and requires re-translation for every execution
- Interpreter could not proceed even with minor errors, thus reporting errors one by one for correction
- The interpretation requires large amount of time, thus reducing the speed of execution
- Programs are executed line by line as soon as they are found error free



Other Translators

- Assembler
  - Assembler translates assembly language program, which uses mnemonic names for operation codes and data addresses into the machine language
- Macro processor
  - A macro processor is a system program that replaces the macro calls in an assembly language program by the macro definition
  - During this text replacement process, the formal arguments in the macro definition are converted into the actual arguments in the macro call.
- Preprocessor
  - A translator whose source language and object language are high level language is termed as preprocessor
- Linker
  - The tool that merges the object files produced by separate compilation and creates single executable module
  - Resolves symbolic references
- Loader
  - Allocates the space for program in the memory
  - Physically places all the machine instructions and data into the memory
  - Loader prepares the loaded modules for execution
  - Different types are
    - Compile and go loader
    - Absolute loader
    - Direct linking loader
    - Relocating loader
    - Dynamic linking loader

Structure of a Compiler

- The goal of compilation is to maps a source program into a semantically equivalent target program and this process has two parts

Analysis (Front End)

- The analysis part validates the program constructs using grammatical structure on them
- If the analysis part detects errors syntactically or semantically, then it must provide informative messages for the user to take corrective action
- Creates an intermediate representation of the source program
- Collects information about the source program and stores it in a data structure called a symbol table, which is passed along with the intermediate representation to the synthesis part

Synthesis (Back End)

- The synthesis generates the desired target program from the intermediate representation and the information in the symbol table

Phases

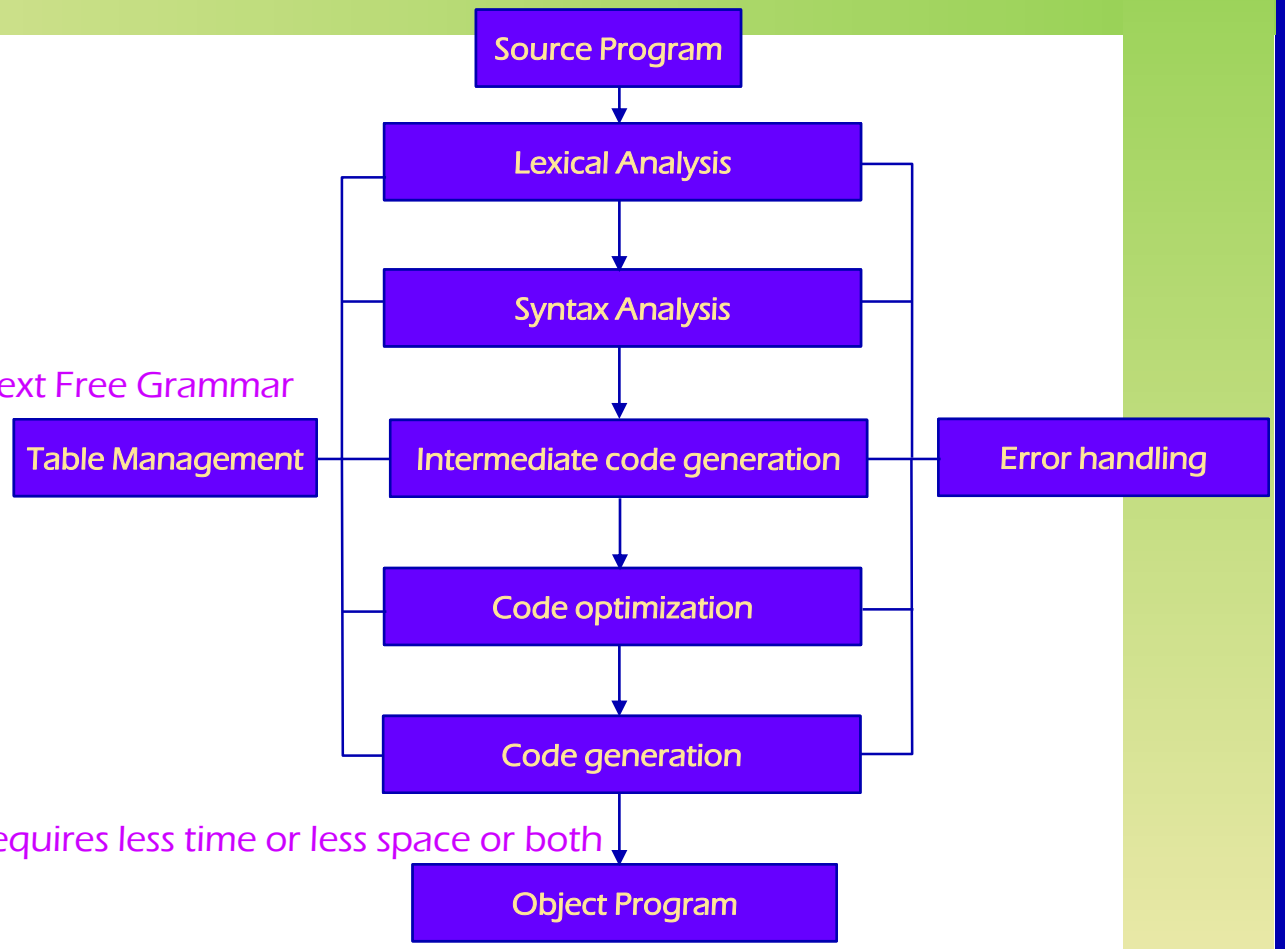
- Highly cohesive functionality which transforms one representation of the source program to another
- Several phases may be grouped together, and the intermediate representations between the grouped phases need not be constructed explicitly

Passes

- Activities from several phases may be grouped together into a pass that reads an input file and writes an output file

Phases of Compiler

- Lexical Analysis
  - Character by character reading, most time consuming phase
  - Tokenization with regular expressions
- Syntax Analysis
  - Groups tokens as per the specification of the language in Context Free Grammar
  - Creates derivation tree for error free programs
- Intermediate Code Generation
  - Constructs intermediate language
  - Uses Syntax Directed Translation Engine
- Code Optimization
  - Transforms program segments into optimal segments, which requires less time or less space or both
  - Data flow/Control flow analysis are required
- Code Generation
  - Transforms intermediate language into object language as per the specification of the compiler
  - Machine dependent and requires the characteristics of the target machine



Output for Phases of Compilation

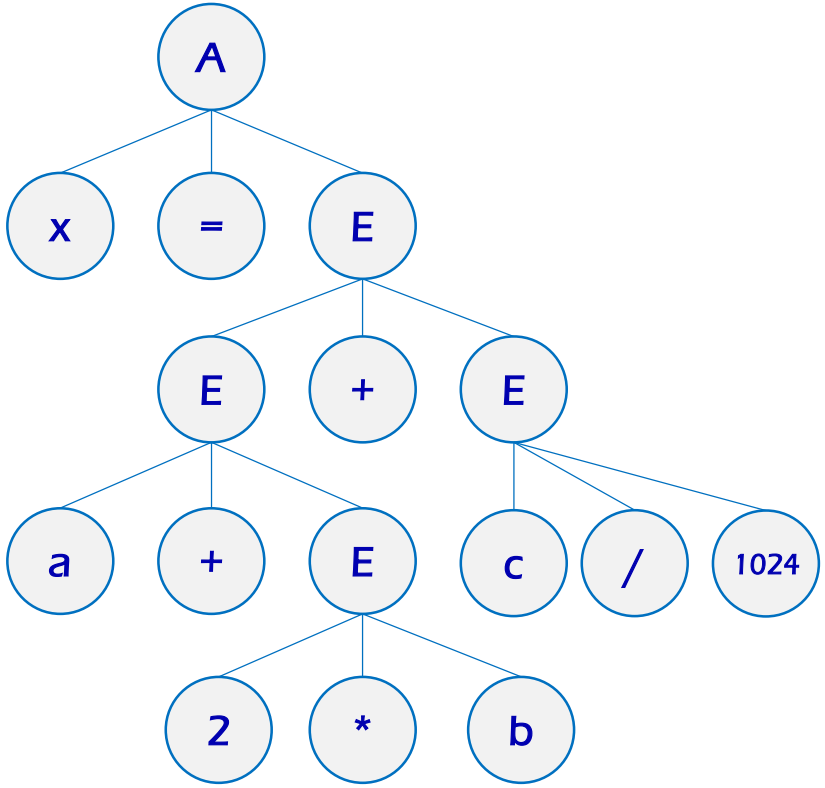
Source Program

x = a + 2 * b + c / 1024

Lexical Analyzer

(id, 16) (op, =) (id, 20)
(op, +) (const, 2) (op, x)
(id, 24) (op, +) (id, 28)
(op, /) (const, 1024)

Syntax Analyzer



Intermediate Code

t1 = 2 * b
t2 = a + t1
t3 = c / 1024
t4 = t2 + t3
x = t4

Code Optimizer

t1 = b + b
t2 = a + t1
t3 = c << 10
t4 = t2 + t3
x = t4

Code Generation

Load b, r1; Add b, r1;
Load a, r2; Add r2, r1;
Load c, r2; Shr 10, r2;
Add r1, r2; Store r2, x;

Compiler Construction Tools

- Use specialized languages for specifying and implementing specific components of compilation
- Hide the details of the components design and produce components that can be easily integrated into the remainder of the compiler
- Familiar Compiler Construction Tools are
  - Parser generators - automatically produce syntax analyzers from a grammatical description of a programming language
  - Scanner generators - produce lexical analyzers from a regular-expression description of the tokens of a language
  - Syntax-directed translation engines - produce collections of routines for walking a parse tree and generating intermediate code
  - Code-generator generators - produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine
  - Data-flow analysis engines - facilitate the gathering of information about how values are transmitted from one part of a program to each other part
  - Compiler-construction toolkits - provide an integrated set of routines for constructing various phases of a compiler

Bootstrapping of Compilers

How the compilers are evolving?

A - Language for machine A
L - Language for which compiler is to be written
S - Subset of features from L

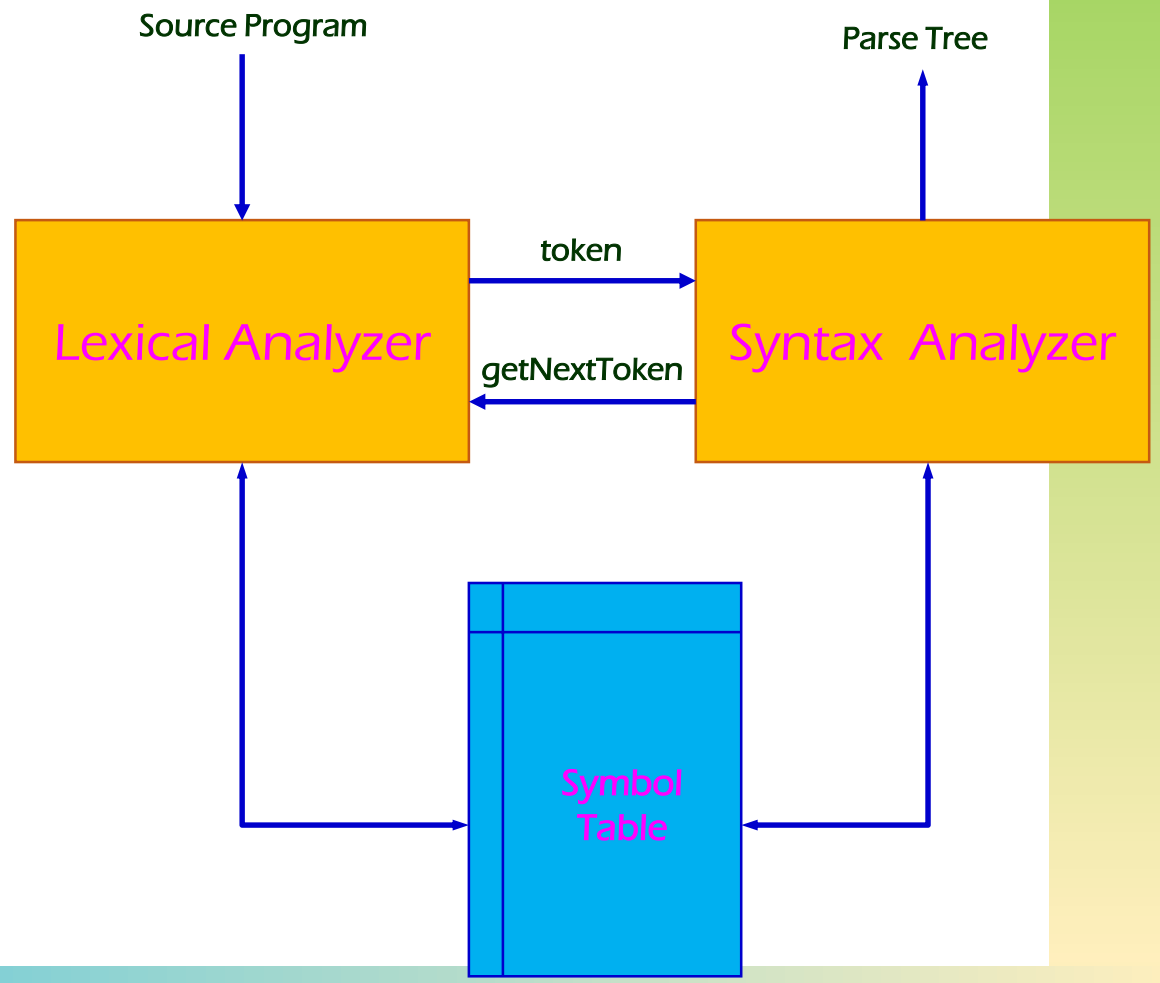
C_Z^{XY} - Program C written in Z for translating X to Y

Lexical Analyzer

First Phase in the Front End of the Compiler

Role of lexical analyzer

- Reads the symbols from the source program, group them into lexemes, produces a sequence of tokens for each lexeme and sent to the parser for syntax analysis
- Interacts with the symbol table for every lexeme constituting an identifier
- Interacts with syntax analyzer as a co-routine in determining the proper token it must pass to the syntax analyzer
- Strips out comments and whitespace (blank, newline, tab, and perhaps other characters that are used to separate tokens in the input)
- Errors will be reported when the lexical analyzer fails to find a valid token
- Separating lexical and syntactic phases results in effective language design
- Simple recognizers can be used for tokenizing and parsing



Lexemes, Patterns, and Tokens

Lexeme

- a sequence of characters in the source program that matches the pattern for a token

Pattern

- a description of the form that the lexemes of a token may take

Token

- A notation representing a kind of lexical unit and returned as a pair consisting of a token name and an optional attribute value

if val1 < val2 then max = val2 else max = val1

if	if	IF
val1	l(l+d)*	(ID, p1)
<	<	LT
val2	l(l+d)*	(ID, p2)
then	then	THEN
max	l(l+d)*	(ID, p3)
=	=	ASSIGN
val2	l(l+d)*	(ID, p2)
else	else	ELSE
max	l(l+d)*	(ID, p3)
=	=	ASSIGN
val1	l(l+d)*	(ID, p1)

Regular Expression for Regular Set

- A regular expression is a language defining notation consisting of finite string of symbols (operands and operators)
- Basis / Primitive cases
  - ϕ is a regular expression denoting the language ϕ, called empty language
  - $\in$ is a regular expression denoting the language $\{\in\}$, language containing empty string only
  - If *a* is any symbol, then *a* is a regular expression denoting the language {a}
- Let R and S be any two regular expressions
  - (R) be the regular expression denoting the same language as L(R)
- R + S is a regular expression denoting the union of L(R) and L(S)

  $L(R+S) = L(R) \cup L(S)$, i.e. either a string from L(R) or a string from L(S)
- RS is a regular expression denoting the language obtained from the concatenation of L(R) and L(S)

  L(RS) = L(R).L(S), i.e. a string from L(R) and a string from L(S) in the same order
- R^* is a regular expression, denoting the closure of L(R)

  $L(R^*) = (L(R))^*$, i.e. the concatenation of any string from L(R) zero or any number of times

Regular Expression : Examples

Regular Expression	Language Recognized	Algebraic Laws
a|b	{ a, b }	r|s = s|r
(a|b)(a|b)	{ aa, a, ba, bb }	r|(s|t) = (r|s)|t
a^*	{ ∈, a, aa, aaa, . . . }	r(st) = (rs)t
$(a\|b)^*$ or $(a^*b^*)^*$	{ ∈, a, b, aa, ab, ba, bb, aaa, . . . }	r(s|t) = rs|rt
$a\|a^*b$	{ a, b, ab, aab, aaab, . . . }	(s|t)r = sr|tr
a?	{ ∈, a }	∈r = r∈ = r
a^+	{ a, aa, aaa, . . . }	$r^+ = rr^*$
$(+\|-)?[0\text{-}9]^+$	Set of all integers	$r^{**} = r^*$
$[a\text{-}z][a\text{-}z0\text{-}9]^*$	Set of all identifiers	$\Phi^* = \in$ $\in^* = \in$

Finite Automata – Recognizer for Regular Set

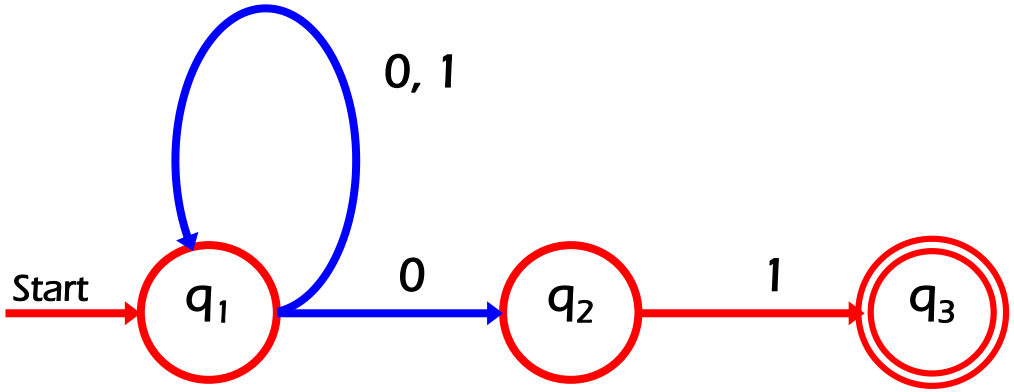
Nondeterministic finite automata (NFA)

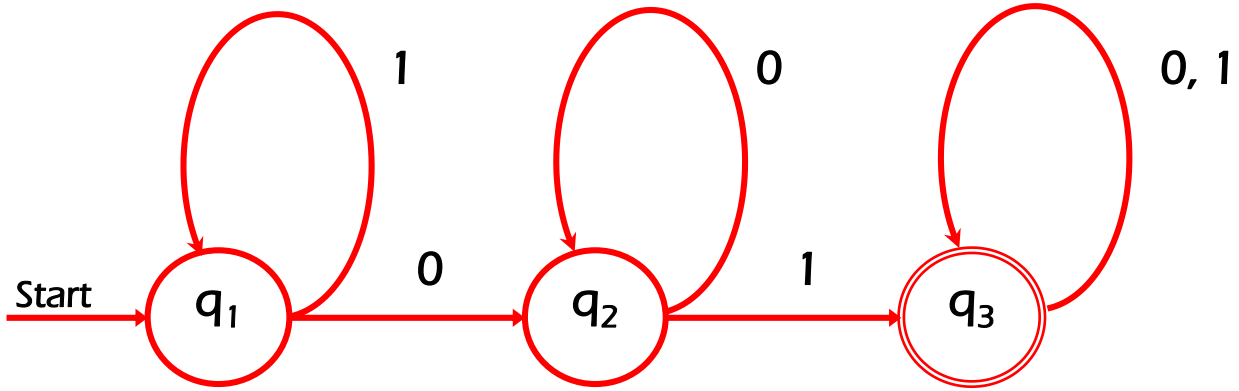
- A finite set of states S
- A set of input symbols C, the input alphabet. We assume that ∈ which stands for the empty string, is never a member of C
- A transition function that gives, for each state, and for each symbol in C U (∈) a set of next states
- A state so from S that is distinguished as the start state (or initial state)
- A set of states F, a subset of S, that is distinguished as the accepting states (or final states)

Deterministic finite automata (DFA)

- There are no moves on input ∈
- For each state *s* and input symbol *a*, there is exactly one edge out of *s* labeled *a*

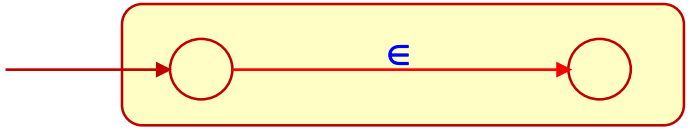
NFA & DFA Examples



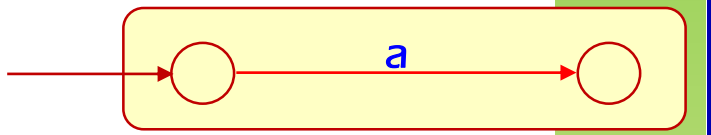


Regular Expression to NFA

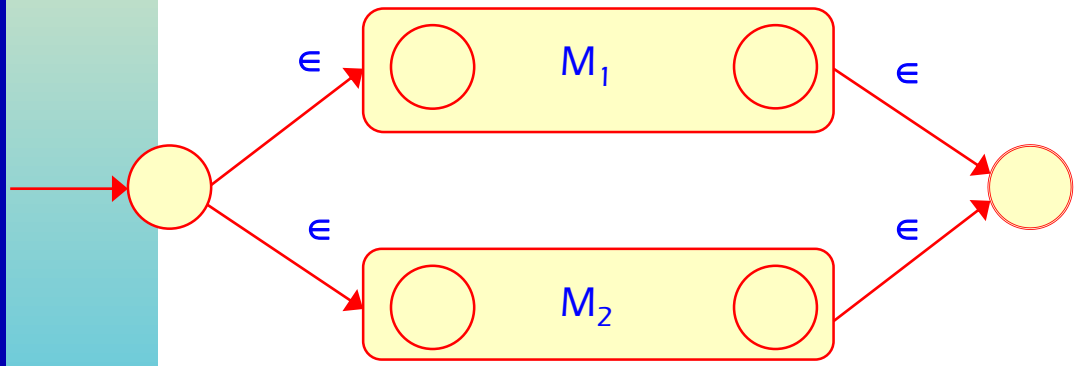
Φ



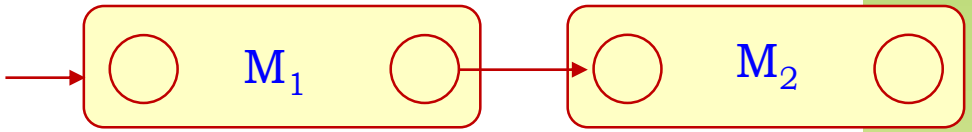
∈



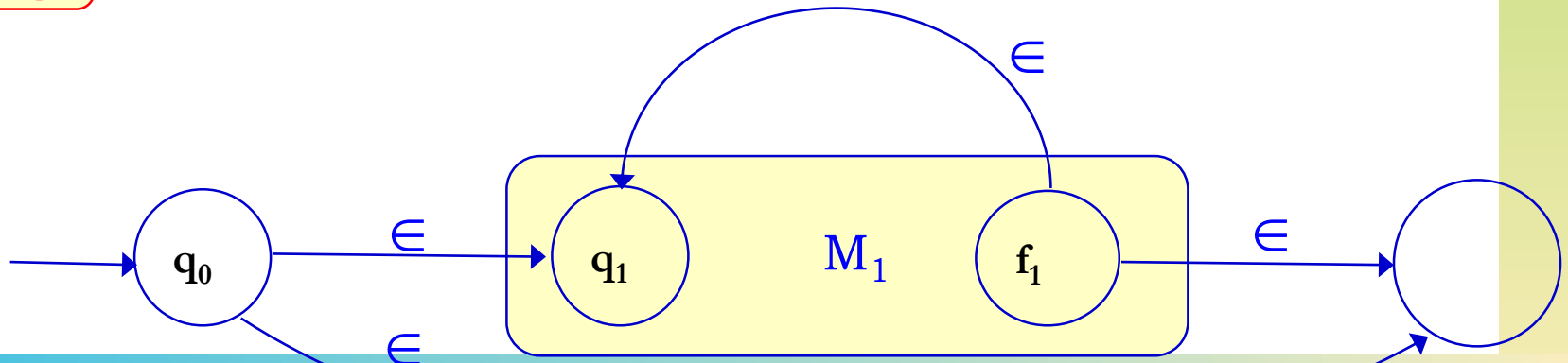
a



$L(M_1)+L(M_2)$

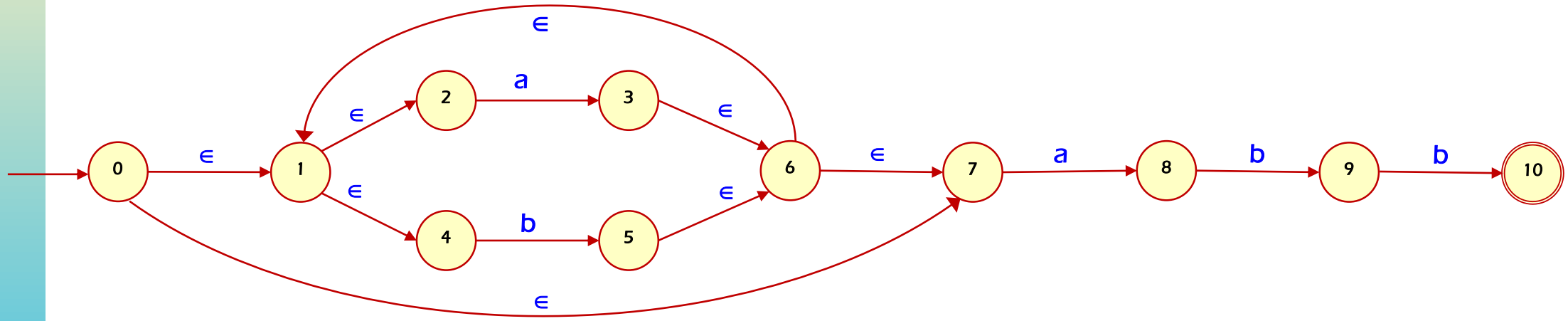


$L(M_1).L(M_2)$



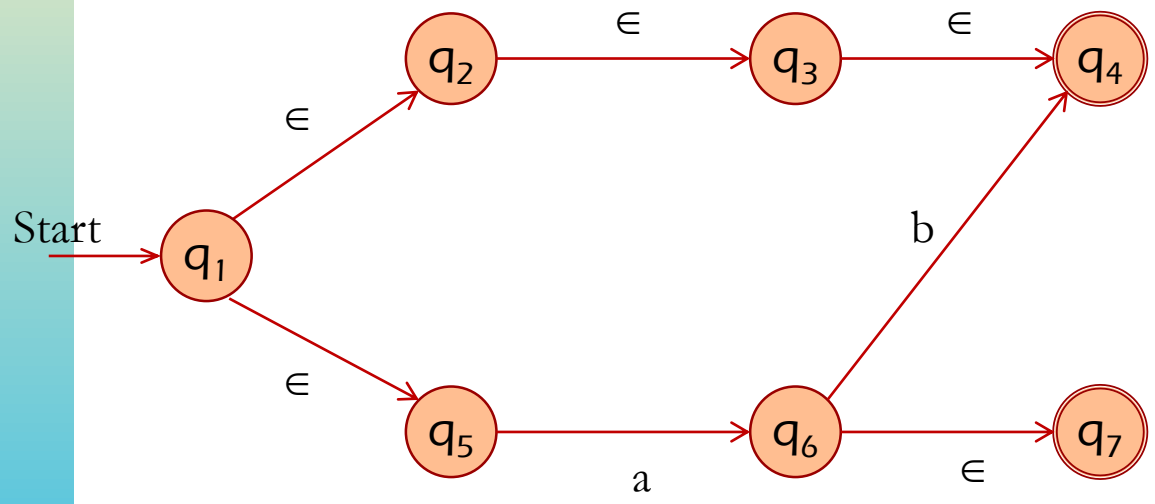
$L(M_1)^*$

NFA for (a|b)*abb



∈ - Closure Computation

∈-Closure (T) - Set of NFA states reachable from NFA states s in T using ∈-transitions alone



∈-Closure($\{q_1\}$) = { q_1, q_2, q_3, q_4, q_5 }

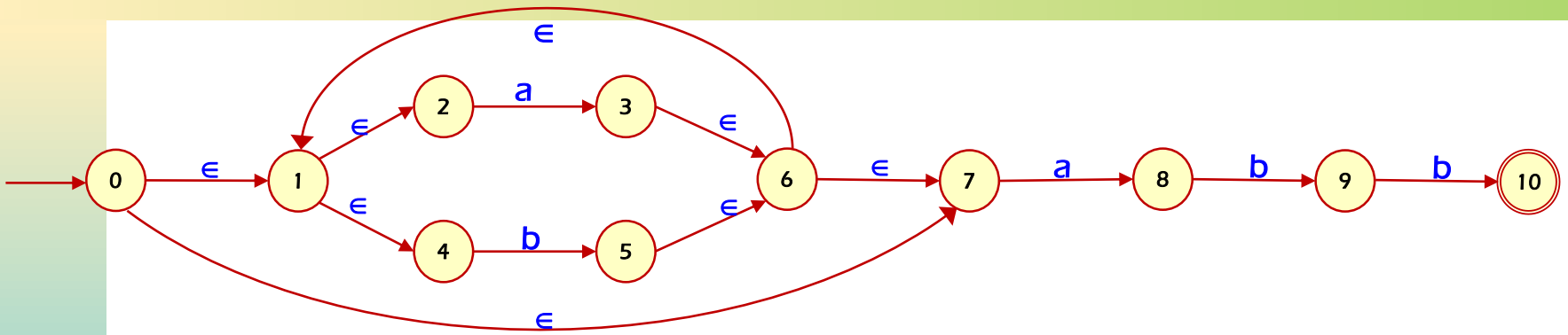
∈-Closure($\{q_2\}$) = { q_2, q_3, q_4 }

∈-Closure($\{q_5\}$) = { q_5 }

∈ - Closure(T) Computation Algorithm

```
push all states of T onto stack;
initialize ∈-closure(T) to T;
while ( stack is not empty )
{   pop t, the top element, off stack;
    for ( each state u with an edge from t
            to u labeled e )
        if ( u is not in ∈-closure(T) )
                { add u to e-closure(T);
                  push u onto stack
                }
}
```

NFA to DFA using subset construction



States	Transitions	
	a	b
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

The Start State ϵ-CLOSURE ({ 0 }) = { 0, 1, 2, 4, 7 } = A

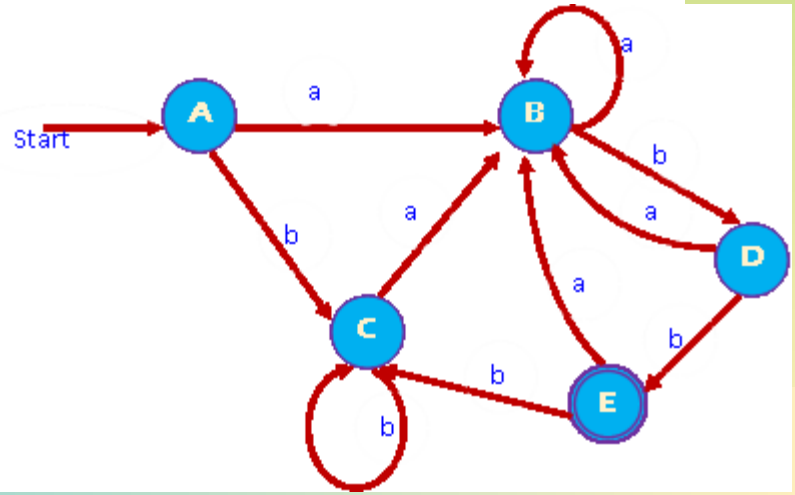
Move(A, a) = { 3, 8 } ϵ-CLOSURE ({ 3, 8 }) = { 3, 8, 6, 7, 1, 2, 4 } = B
Move(A, b) = { 5 } ϵ-CLOSURE ({ 5 }) = { 5, 6, 7, 1, 2, 4 } = C

Move(B, a) = { 3, 8 } ϵ-CLOSURE ({ 3, 8 }) = { 3, 8, 6, 7, 1, 2, 4 } = B
Move(B, b) = { 5, 9 } ϵ-CLOSURE ({ 5, 9 }) = { 5, 9, 6, 7, 1, 2, 4 } = D

Move(C, a) = { 3, 8 } ϵ-CLOSURE ({ 3, 8 }) = { 3, 8, 6, 7, 1, 2, 4 } = B
Move(C, b) = { 5 } ϵ-CLOSURE ({ 5 }) = { 5, 9, 6, 7, 1, 2, 4 } = C

Move(D, a) = { 3, 8 } ϵ-CLOSURE ({ 3, 8 }) = { 3, 8, 6, 7, 1, 2, 4 } = B
Move(D, b) = { 5, 10} ϵ-CLOSURE ({ 5, 10 }) = { 5, 10, 6, 7, 1, 2, 4 } = E

Move(E, a) = { 3, 8 } ϵ-CLOSURE ({ 3, 8 }) = { 3, 8, 6, 7, 1, 2, 4 } = B
Move(E, b) = { 5 } ϵ-CLOSURE ({ 5 }) = { 5, 6, 7, 1, 2, 4 } = C



Minimizing the number of states of a DFA

1. Initialize the partition P with two groups, the final states and the set of non-final states. $\pi = (F, S-F)$
2. Construct π_{new}, a refinement of π consists of groups of π, each split into one or more pieces
3. If $\pi = \pi_{new}$ then Continue from 4, otherwise assign π_{new} to π and repeat steps 2 and 3
4. Choose one representative from each group of π and becomes the states in the minimized automata M'

   Let s be a representative state, s has a transition on input 'a' to the state 't' in M

   Let 'r' be the representative in the group where 't' is a member state

   Then in M' there is a transition from 's' to 'r' for input symbol 'a'

5. Initial state of M' is the representative of the group containing s_0 in M
6. The final states of M' are the representatives chosen from F
7. Remove the dead states, which are non-final states with transitions to itself for all input symbols
8. Remove the states unreachable from initial state.

Procedure for π_{new} construction:

For each group G of π do

- Partition G into subgroups such that two states s and t of G are in the same sub-group if and only if for all input symbols a, states s and t have transitions to states in the same group of π.
- Place all subgroups so formed in π_{new}

Minimization Example

States	Transitions	
	a	b
→A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

π = ((E) , (A, B, C, D))

The transitions from A, B, C for all inputs lead to the same group. But the transition from D for input symbol 'a' leads to the first group while the transition for input 'b' leads to second group.

Hence it is required to split the group (A, B, C, D) to two groups as (A, B, C) and (D)

π_{new} = ((E) (A, B, C) (D))

Since π and π_{new} are not equal assign π_{new} to π and construct π_{new} .

π = ((E) (D) (A, B, C)))

The transitions from A, C leads to same group but the transitions from B go to different groups

Thus (A, B, C) is divided as (A, C), (B).

π_{new} = ((E) (D) (A, C) (B))

Since π and π_{new} are not equal assign π_{new} to π and construct π_{new} .

π = ((E) (D) (A, C) (B))

In π, only one group has more than one state and transitions from A, C lead to same groups. Hence that group need not be divided and π and π_{new} will be equal

Now choose one representative from each group. The representatives are (A, B, D, E), the states of the minimized DFA.

States	Transitions	
	a	b
→A	B	A
B	B	D
D	B	E
*E	B	A

Scanner Design using FLEX