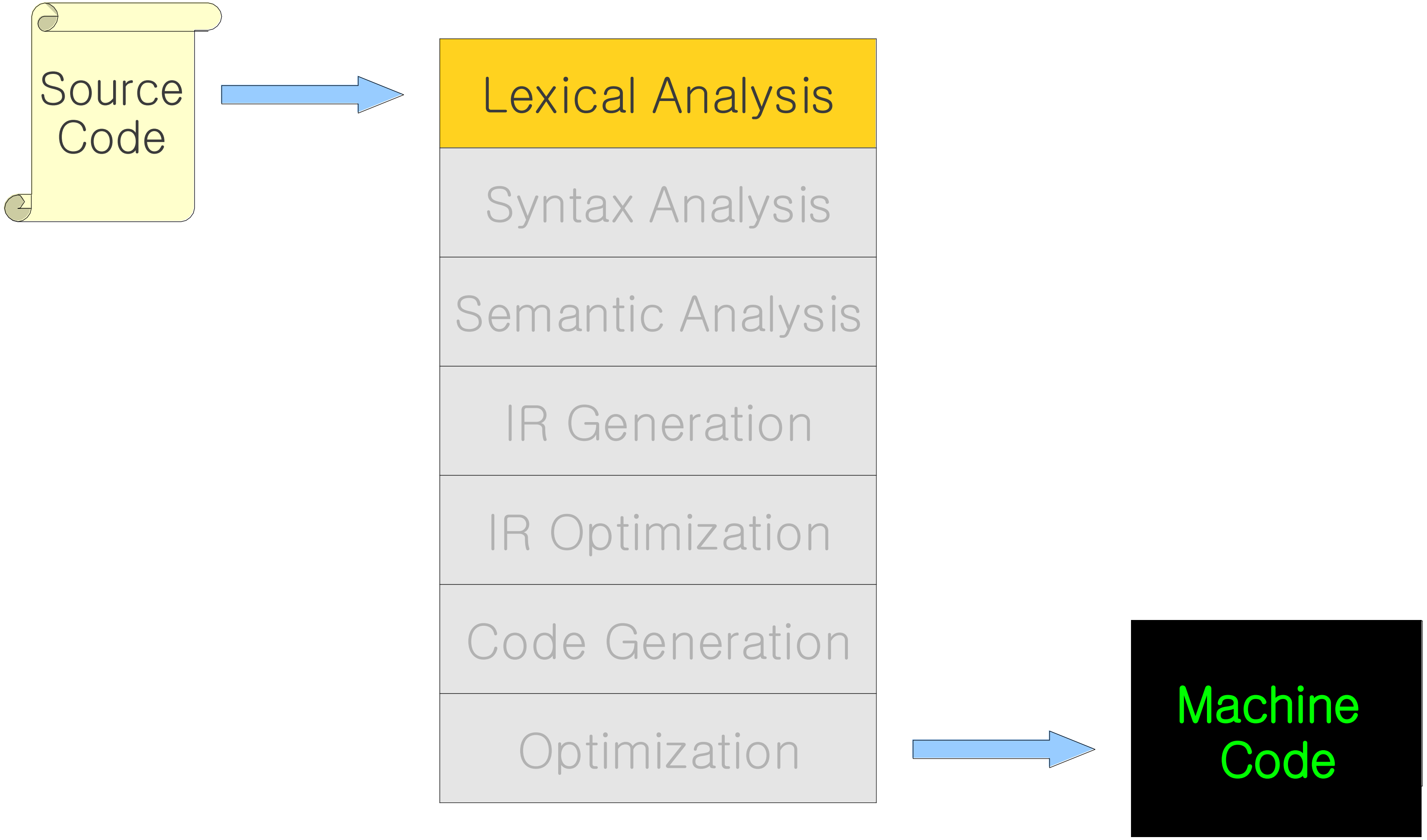
Compiler
- 2-1. Lexical Analysis -

JIEUNG KIM

jieungkim@yonsei.ac.kr

YONSEI UNIVERSITY

Where are we?


Source Code
Lexical Analysis
Syntax Analysis
Semantic Analysis
IR Generation
IR Optimization
Code Generation
Optimization
Machine Code

Outlines

- Basic concepts of formal grammars
- Role of the lexical analyzer
- Choose a token
- Finite automata
- Regular expression
- Specification of tokens
- Recognition of tokens
- Error handling
- Challenges in scanning
- Lex : lexical analyzer generator

YONSEI UNIVERSITY

Basic concepts of formal grammar

Basic concepts of formal grammar

- Programming language specs
  - Since the 1960s, the syntax of every significant programming language has been specified by a formal grammar
  - First done in 1959 with BNF (**Backus-Naur Form** or **Backus-Normal Form**) used to specify the syntax of ALGOL 60
  - Borrowed from the linguistics community (Chomsky)

Basic concepts of formal grammar

- Example of grammar for a tiny language

```
program ::= statement | program statement
statement ::= assignStmt | ifStmt
assignStmt ::= id = expr ;
ifStmt ::= if ( expr ) stmt
expr ::= id | int | expr + expr
id ::= a | b | c | i | j | k | n | x | y | z
int ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

YONSEI UNIVERSITY

Basic concepts of formal grammar

- **Productions**
  - The rules of a grammar are called productions
  - Rules contain
    - Nonterminal symbols: grammar variables (program, statement, id, etc.).
    - Terminal symbols: concrete syntax that appears in programs
      (a, b, c, 0, 1, if, (,), ⋯)
  - Meaning of "**nonterminal ::= <sequence of terminals and nonterminals>**"
    - In a derivation, an instance of *nonterminal* can be replaced by the sequence of *terminals and nonterminals* on the right of the production
  - Often, there are two or more productions for one nonterminal – use any in different parts of derivation

Basic concepts of formal grammar

- Alternative notations
  - There are several syntax notations for productions in common use; all mean the same thing

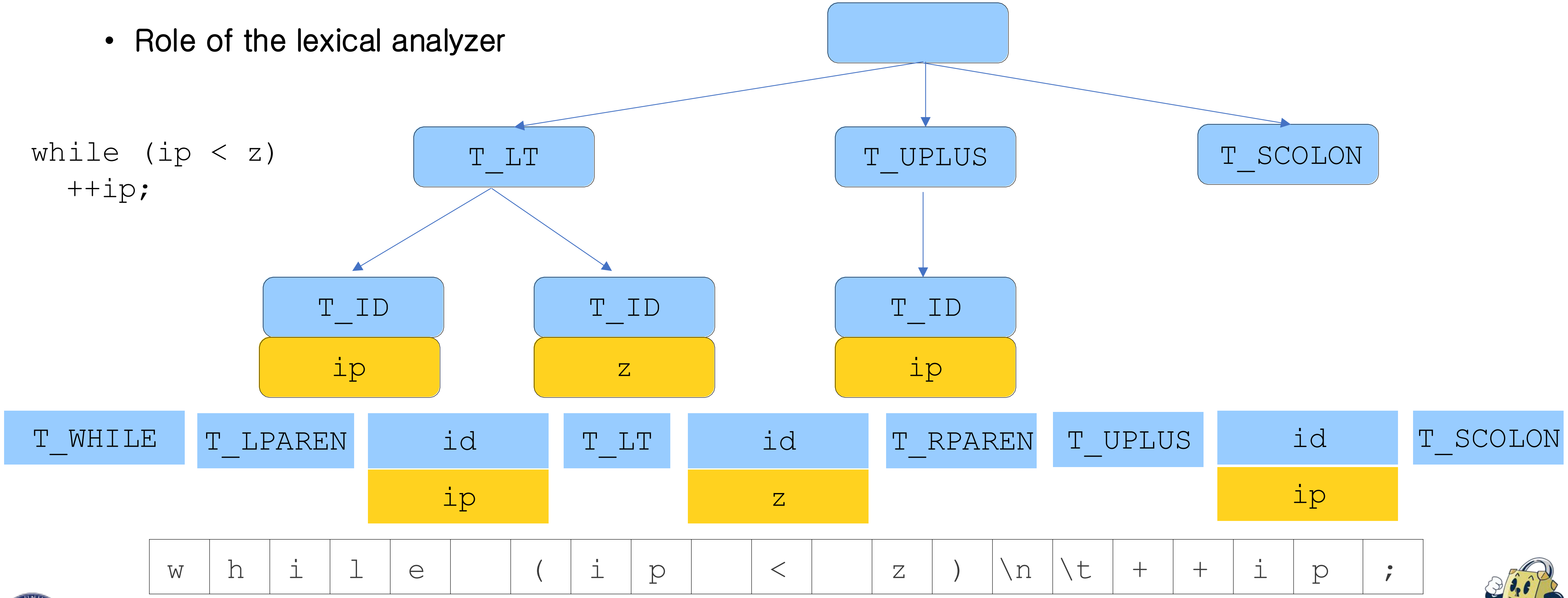
```
ifStmt ::= if ( expr ) stmt
ifStmt → if ( expr ) stmt
<ifStmt> ::= if ( <expr> ) <stmt>
```

Role of the lexical analyzer

Role of the lexical analyzer

- Role of the lexical analyzer

```
while (ip < z)
  ++ip;
```



Role of the lexical analyzer

- Role of the lexical analyzer

```
do[for] = new 0;
```

T_DO	T_LBRAKET	T_FOR	T_RBRAKET	T_ASSIGN	T_NEW	T_NUM	T_SCOLON
						0	

d	o	[	f	o	r	]		=		n	e	w		0	;

YONSEI UNIVERSITY

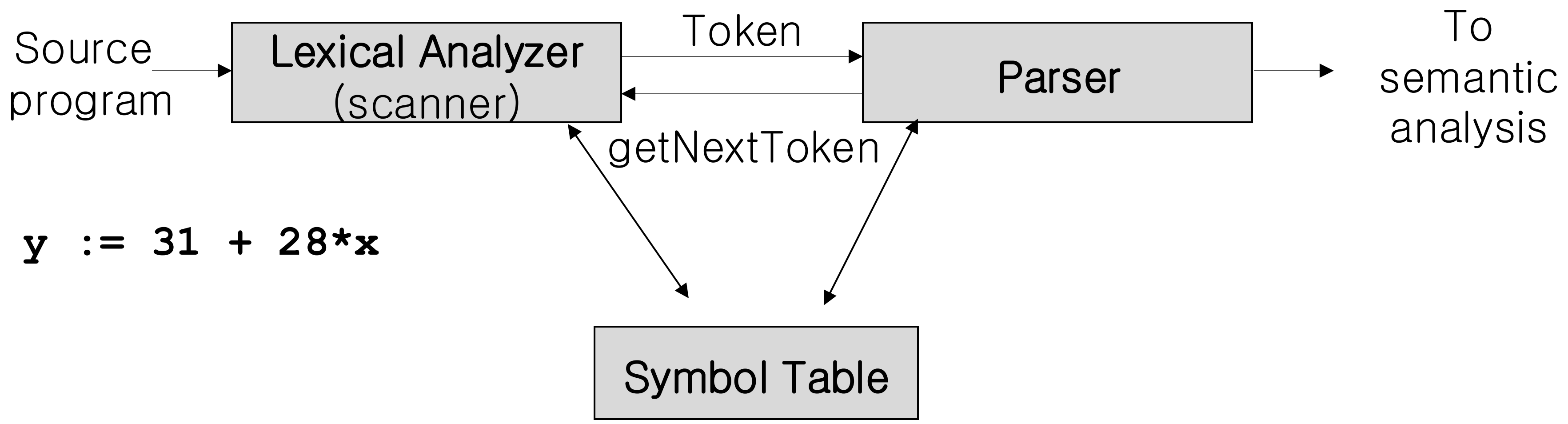
Role of the lexical analyzer

- **Parsing and lexing**
  - Parsing
    - Parsing: Reconstruct the derivation (syntactic structure) of a program.
    - In principle, a single recognizer could work directly from a concrete, character-by-character grammar.
    - In practice this is never done.
  - Lexing and parsing
    - In real compilers the recognizer is split into two phases.
      - Lexical analyzer (scanner): Translate input characters to tokens.
        - Also, report lexical errors like illegal characters and illegal symbols.
      - Parser: Read token stream and reconstruct the derivation.

Role of the lexical analyzer

- Role of the lexical analyzer
  - Read source code and generate token.

<**T_ID,** “**y**”> <**T_ASSIGN**> <**T_NUM,** 31> <**T_PLUS**> <**T_NUM,** 28> <**T_MUL**> <**T_ID,** “**x**”>



Role of the lexical analyzer

- **Minor roles of lexical analyzer**
  - Removal of white space and comments
  - Reading ahead
    - Need to read ahead some characters before its decision as a token.
      - E.g., > and >=
    - Maintain input buffers (fetching character and push back it to the buffer)
  - Management of symbol table
    - Symbol table generation
    - ID insertion/referencing
  - Handling constants
    - Lexical analyzer collects characters to generate integers and compute collaborative numerical value.
    - In parser, translation numbers can be treated as single unit
      <T_NUM, 31> <T_PLUS> < T_NUM,28>

Role of the lexical analyzer

- **Why do we divide two part in a compiler**
  - Simplicity of design
    - May requires multiple tokens to parse in parser
  - Improved compiler efficiency
    - Lexical analysis is rather simpler than parser (even automate implementation)
  - Compiler portability
    - Different languages have different tokens (symbols)

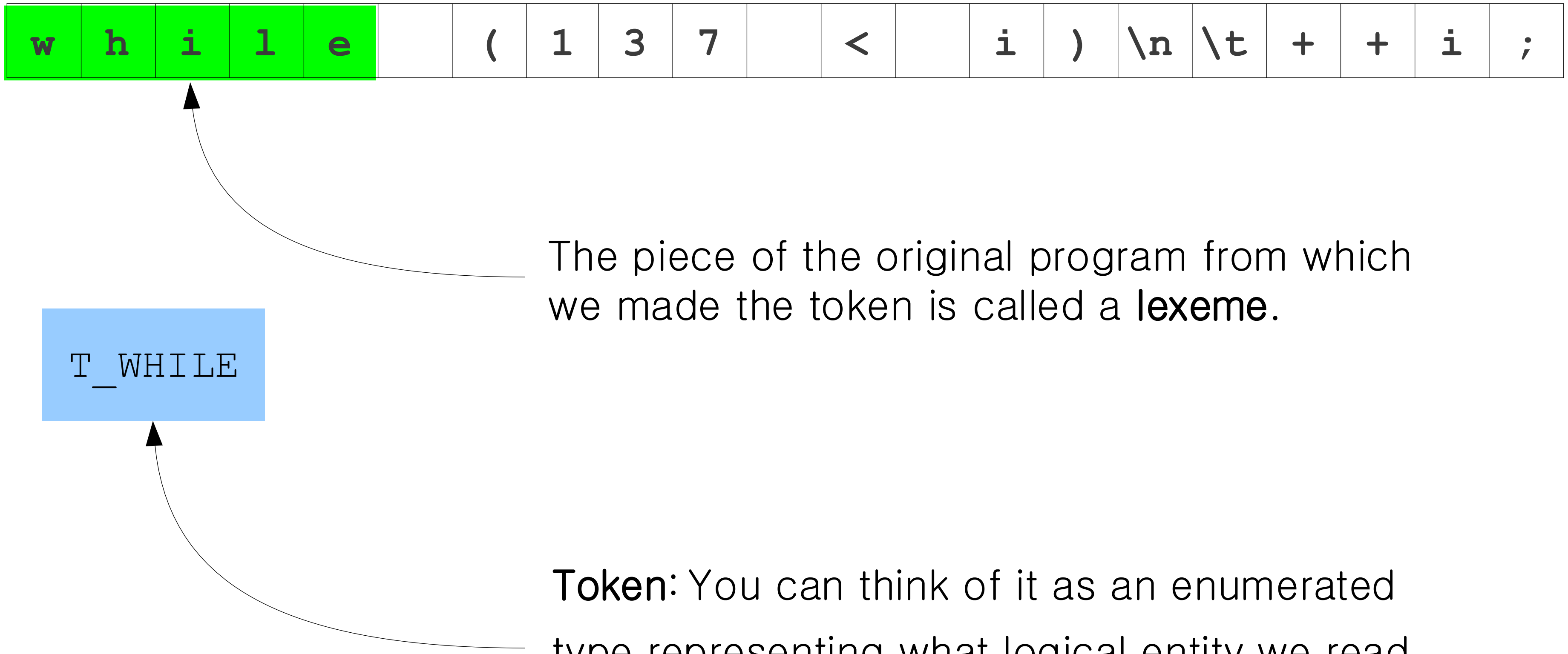
Role of the lexical analyzer

- Lexing a file

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

Role of the lexical analyzer

- Lexing a file



The piece of the original program from which we made the token is called a **lexeme**.

Token: You can think of it as an enumerated type representing what logical entity we read out of the source code.

Role of the lexical analyzer

- Lexing a file

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_WHILE

Sometimes we will discard a lexeme rather than storing it for later use. Here, we ignore whitespace, since it has no bearing on the meaning of the program.

Role of the lexical analyzer

- Lexing a file

w	h	i	l	e		(	1	3	7		<		i	)	\n	\t	+	+	i	;

T_WHILE	T_LPAREN	T_INT
		137

Some tokens can have **attributes** that store extra information about the token. Here we store which integer is represented.

Role of the lexical analyzer

- Token (lexical token)
  - A string with an assigned and identified meaning
  - Structured as a pair consisting of a token name and an optional token value (some tokens may have attributes)
    - Examples: integer constant token will have the actual integer (17, 42, …) as an attribute; identifiers will have a string with the actual id
  - The token name is a category of lexical unit in the grammar

YONSEI UNIVERSITY

Role of the lexical analyzer – self-study page

- **Lexing a file example**
  - Input text

```
if (x >= y) y = 42;
```

  - Token Stream

```
<T_IF>  <T_LPAREN>   <T_ID, x>   <T_GE>   <T_ID, y>
<T_RPAREN>  <T_ID, y>  <T_ASSIGN>  <T_INT, 42>  <T_SCOLON>
```

Choose a token

Choose a token

- What tokens are useful here?

```
for (int k = 0; k < myArray[5]; ++k) {
  cout << k << endl;
}
```

```
for        {
int        }
<<         ;
=          <
(          [
)          ]
++
```

Choose a token

- Choosing good tokens
  - Very much dependent on the language
  - Typically
    - Give keywords their own tokens.
    - Give different punctuation symbols their own tokens.
    - Group lexemes representing identifiers, numeric constants, strings, etc. into their own groups.
    - Discard irrelevant information (whitespace, comments)

YONSEI UNIVERSITY

Choose a token

- Choosing tokens – example
  - FORTRAN
    - Whitespace is irrelevant.
    - Can be difficult to tell when to partition input.

```
DO 5 I = 1.25
DO5I = 1.25
```

  - C++
    - Nested template declarations
    - Again, can be difficult to determine where to split.

```
vector < vector < int >> myVector
(vector < (vector < (int >> myVector)))
```

  - PL/1
    - Keywords can be used as identifiers.
    - Can be difficult to determine how to label lexemes.

```
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
IF THEN THEN THEN = ELSE; ELSE ELSE = IF
```

YONSEI UNIVERSITY

Choose a token

- Challenges in scanning
  - How do we determine which lexemes are associated with each token?
  - When there are multiple ways we could scan the input, how do we know which one to pick?
  - How do we address these concerns efficiently?

Associating lexemes with tokens

Choose a token

- **Lexemes and tokens**
  - Tokens give a way to categorize lexemes by what information they provide.
  - Some tokens might be associated with only a single lexeme:
    - Tokens for keywords like **if** and **while** probably only match those lexemes exactly.
  - Some tokens might be associated with lots of different lexemes:
    - All variable names, all possible numbers, all possible strings, etc.

Choose a token – self-study page

- Typical tokens in programming languages
  - Operators & punctuation:
    - + - * / ( ) { } [ ] ; : :: < <= == = != ! …
    - Each of these is a distinct lexical class
  - Keywords
    - if while for goto return switch void …
    - Each of these is also a distinct lexical class (not a string)
  - Identifiers
    - A single ID lexical class, but parameterized by actual id
  - Integer constants
    - A single INT lexical class, but parameterized by int value
  - Other constants, etc.

Choose a token – self-study page

- **Token, pattern, and lexeme**
  - Token: <token_name, optional attributes>
    - token_name: lexical aunit (e.g., a keyword) or character denoting ID
  - Pattern: description of lexemes
  - Lexeme: a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an instance of that token

Token	Informal description	Sample lexemes
`T_IF`	characters i, f	`if`
`T_ELSE`	Characters e, l, s, e	`else`
`T_COMP`	< or > or <= or >= or == or !=	`<=, !=`
`T_ID`	letter followed by letters and digits	`pi, score, D2`
`T_NUM`	any numeric constant	`3.14159, 0, 6, 02e23`
`T_STRING`	Anything but “, surrounded by “’s	`“core dumped”`

Choose a token

- **Goals of lexical analysis**
  - Convert from physical description of a program into sequence of of **tokens**.
    - Each token represents one logical piece of the source file – a keyword, the name of a variable, etc.
  - Each token is associated with a **lexeme**.
    - The actual text of the token: "137," "int," etc.
  - Each token may have optional **attributes**.
    - Extra information derived from the text – perhaps a numeric value.
  - The token sequence will be used in the parser to recover the program structure.

YONSEI UNIVERSITY

Choose a token

- Set of lexemes
  - Idea: Associate a set of lexemes with each token.
  - We might associate the “number” token with the set { 0, 1, 2, …, 10, 11, 12, … }
  - We might associate the “string” token with the set { "", "a", "b", "c", … }
  - We might associate the token for the keyword while with the set { while }.

How do we describe which (potentially infinite)
set of lexemes is associated with each token type?

Choose a token

- Formal language
  - A **formal language** is a set of strings.
    - Many infinite languages have finite descriptions:
      - Define the language using an automaton.
      - Define the language using a grammar.
      - Define the language using a regular expression
    - We can use these compact descriptions of the language to define sets of strings.
    - Over the course of this class, we will use all of these approaches.

Choose a token

- Regular expression
  - **Regular expressions** are a family of descriptions that can be used to capture certain languages (the regular languages).
  - Often provide a compact and human- readable description of the language.
  - Used as the basis for numerous software systems, including the flex tool we will use in this course.

Choose a token

- Specification of tokens
  - Use **regular expressions** instead of specifying all lexeme patterns (for efficiency)
    - The lexical grammar (structure) of most programming languages can be specified with regular expressions
    - (Sometimes a little cheating is needed)
- Recognition of tokens
  - Tokens can be recognized by a **deterministic finite automaton**
  - Can be either table-driven or built by hand based on lexical grammar
- Lexical analyzer
  - Code that implements deterministic finite automaton
  - Lex: lexical analyzer generator

Choose a token – self-study page

- Principle of longest match
  - In most languages, the scanner should pick the longest possible string to make up the next token if there is a choice
- Example

```
return maybe != iffy;
```

```
<T_RETURN>  <T_ID, maybe>  <T_NEQ>  <T_ID, iffy>  <T_SCOLON>
```

  - Should be recognized as 5 tokens
    - `!=` is one token, not two
    - "`iffy`" is an ID, not <T_IF> followed by <T_ID, fy>

Choose a token – self-study page

- Lexical errors
  - `fi (a == f(x)) …`
    - Lexer cannot decide it's an error or not, why?
      - Case 1: `fi` is spelling miss of `if`
      - Case 2: `fi` is an undeclared function name
  - Let parser handle it

YONSEI UNIVERSITY

Finite automata

Finite automata

- Finite automata (FA)
  - Finite automata are finite collections of states with transition rules that take you from one state to another
  - Original application was sequential switching circuits, where the "state" was the settings of internal bits
  - Today, several kinds of software can be modeled by finite automata

Finite automata

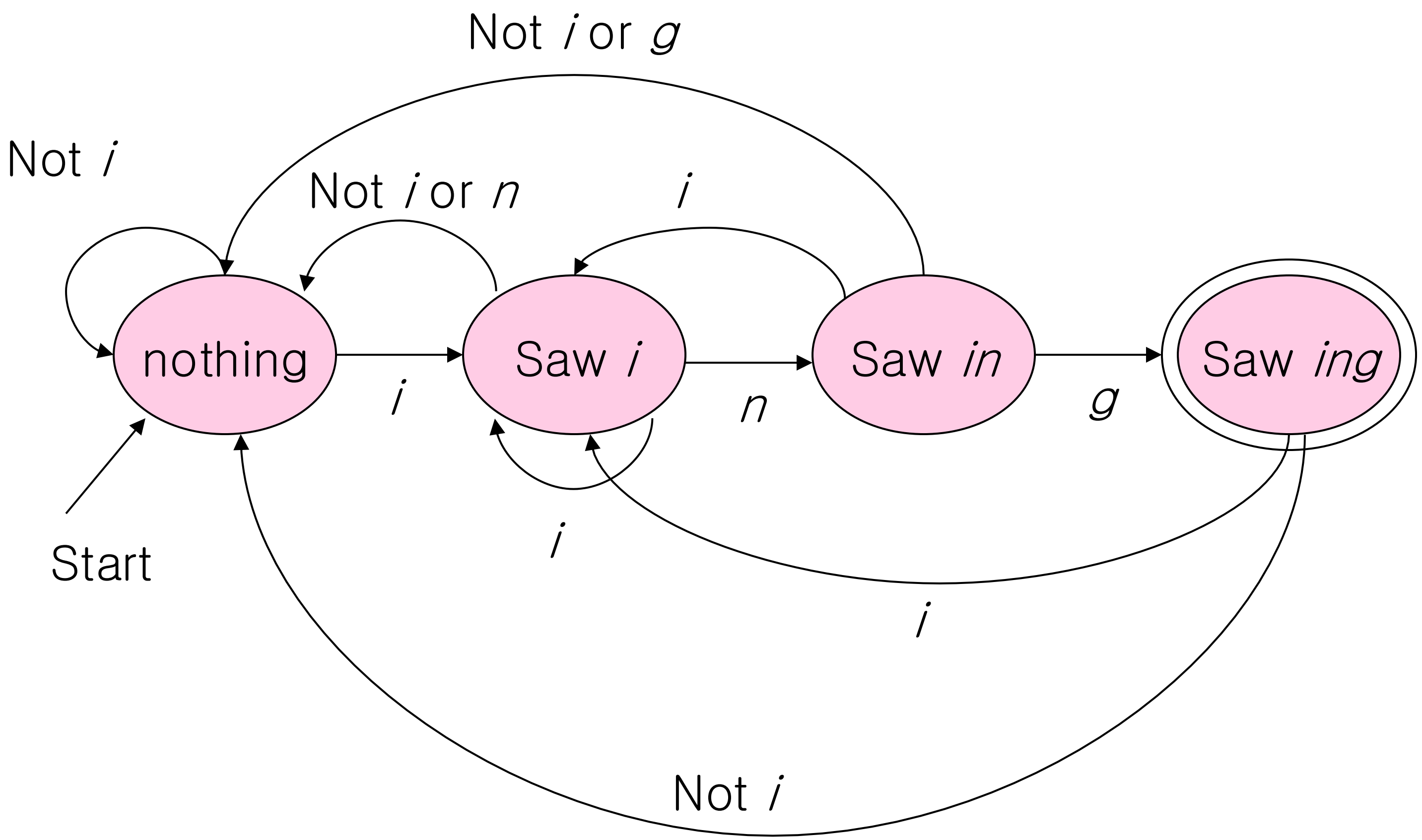
- Finite automata is used as a model for
  - Software for designing digital circuits
  - Lexical analyzer in compiler
  - Text editor searching for keywords in a file or web
  - Software for verifying finiteness such as communication protocols
- Example
  - Modeling on-off switch
  - Recognizing English words
  - Modeling vending machine
  - Etc.

Finite automata

- **Representing finite automata**
  - Simplest representation is often a graph
    - Nodes = states
    - Arcs indicate state transitions
    - Labels on arcs tell what causes the transition

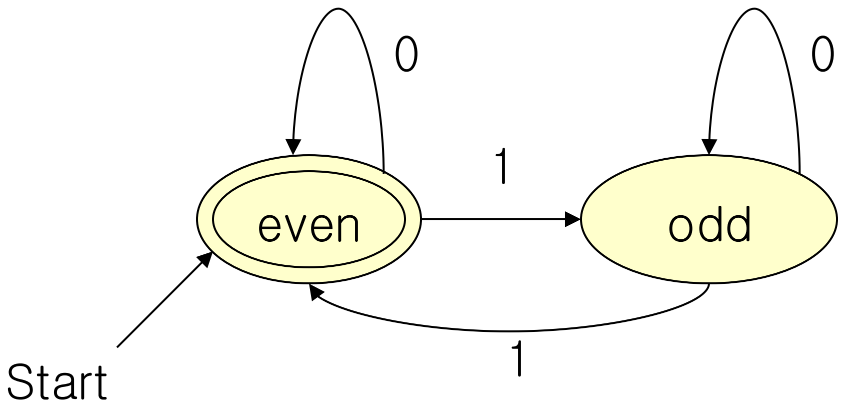
Finite automata

- Example: Recognizing strings ending in "ing"



Finite automata

- Example: An even number of 1's



Finite automata

- General comments related to finite automata
  - Some things are easy with finite automata
    - Substrings (…abcabc…)
    - Subsequences (…a…b…c…b…a…)
    - Modular counting (odd number of 1's)
  - Some things are impossible with finite automata (we will prove this later)
    - An equal number of a's and b's
    - More 0's than 1's
  - But when they **can** be used, they are fast

Language

- Alphabets
  - An *alphabet* is any finite set of symbols (characters)
  - Examples: ASCII, Unicode, {0,1} (*binary alphabet*), {a,b,c}
- Strings
  - The set of *strings* over an alphabet Σ is the set of lists, each element of which is a member of Σ
    - Strings shown with no commas, e.g., abc
  - Σ^* denotes this set of strings
  - $|s|$ denotes the length of string s
  - ε denotes the empty string, thus $|\varepsilon| = 0$

YONSEI UNIVERSITY

Language

- Example: Strings
  - $\{0,1\}^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, . . . \}$
  - Subtlety: 0 as a string, 0 as a symbol look the same
    - Context determines the type

Language

- Languages
  - A *language* is a subset of Σ^* for some alphabet Σ
  - Example: The set of strings of 0's and 1's with no two consecutive 1's
  - L = {ε, 0, 1, 00, 01, 10, 000, 001, 010, 100, 101, 0000, 0001, 0010, 0100, 0101, 1000, 1001, 1010, . . . }

Deterministic finite automata

- Deterministic finite automata
  - A formalism for defining languages, consisting of:
    - A finite set of *states* (Q, typically)
    - An *input alphabet* (Σ, typically)
    - A *transition function* (δ, typically)
    - A *start state* (q_0, in Q, typically)
    - A set of *final states* ($F \subseteq Q$, typically)

YONSEI UNIVERSITY

Deterministic finite automata

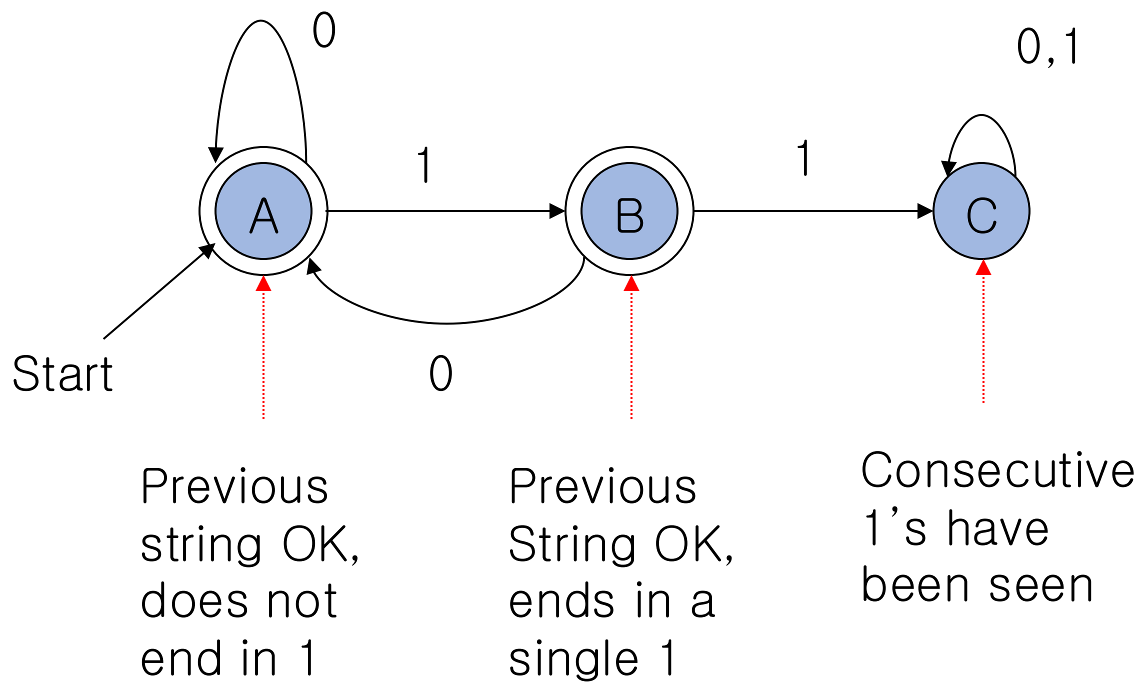
- The transition function
  - Takes two arguments: a state and an input symbol
  - $\delta(q, a)$ = the state that the DFA goes to when it is in state *q* and input *a* is received

Deterministic finite automata

- Graph representation of DFA's
  - Nodes = states
  - Arcs represent transition function
    - Arc from p to q labeled by all those input symbols that have transitions from p to q
  - Arrow labeled "start" to the start state
  - Final states indicated by double circles

Deterministic finite automata

- Example: Graph of a DFA
  - Accepts all strings without two consecutive 1's



Deterministic finite automata

- Example: Transition table of a DFA

Final states starred

Arrow for start state

Columns = input symbols

		0	1
→ *	A	A	B
*	B	A	C
	C	C	C

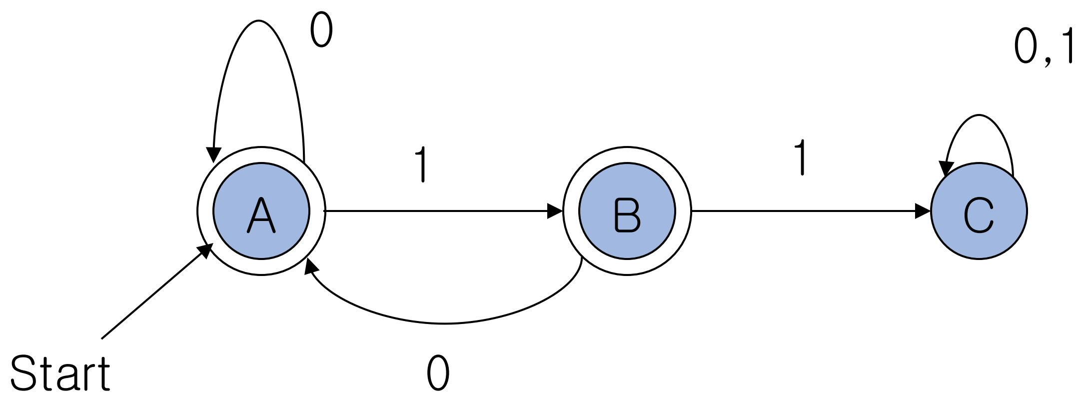
Rows = states

Deterministic finite automata

- Language of a DFA
  - Automata of all kinds define languages
  - If A is an automaton, L(A) is its language
  - For a DFA A, L(A) is the set of strings labeling paths from the start state to a final state
  - Formally: L(A) = the set of strings w such that $\delta(q_0, w)$ is in F

Deterministic finite automata

- **Example: String in a language**
  - String 101 is in the language of the DFA below
    - Start at A
    - Follow arc labeled 1
    - Then arc labeled 0 from current state B
    - Finally arc labeled 1 from current state A (result is an accepting state, so 101 is in the language)



The language of this DFA is:
{w | w is in {0,1}* and w does not have two consecutive 1's}

Deterministic finite automata

- Regular languages
  - A language L is ***regular*** if it is the language accepted by some DFA
    - Note: the DFA must accept only the strings in L, no others
  - Regular Languages can be described in many ways, e.g., regular expressions
  - They appear in many contexts and have many useful properties
  - Examples
    - the strings that represent floating point numbers in your favorite language is a regular language
    - L_3 = { w | w in $\{0,1\}^*$ and w, viewed as a binary integer is divisible by 23}
  - Some languages are not regular
    - Intuitively, regular languages "cannot count" to arbitrarily high integers

Deterministic finite automata

- Example: A nonregular language
  - $L_1 = \{0^n1^n \mid n \geq 1\}$
    - Note: a^i is conventional for i a's
      - Thus, $0^4 = 0000$
    - Read: "The set of strings consisting of n 0's followed by n 1's, such that n is at least 1.
    - Thus, $L_1 = \{01, 0011, 000111, \cdots\}$
  - $L_2 = \{w \mid w \text{ in } \{(,)\}^*$ and w is *balanced* $\}$
    - Note: alphabet consists of the parenthesis symbols '(' and ')'
    - Balanced parenthesis are those that can appear in an arithmetic expression
      - E.g.: (), ()(), (()), (()()),⋯

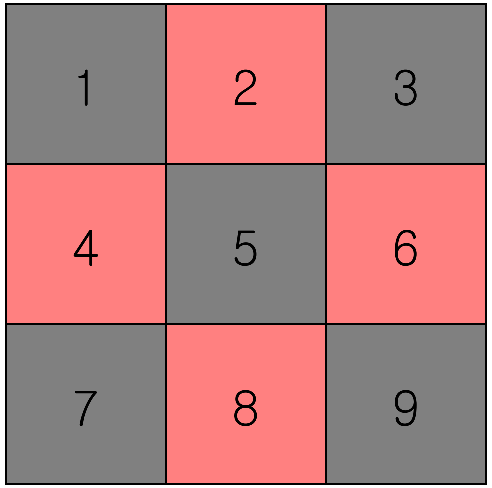
YONSEI UNIVERSITY

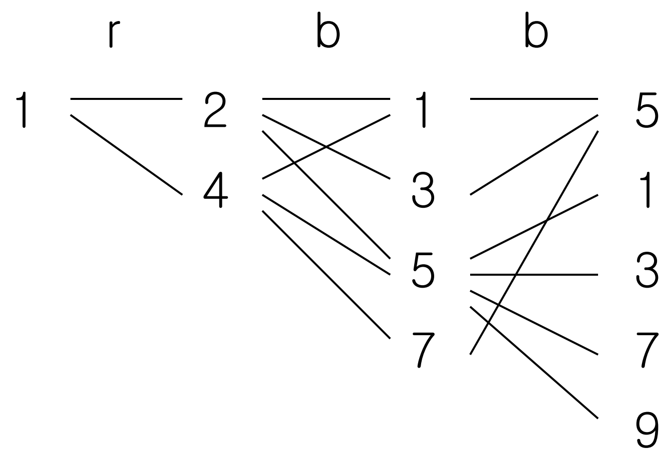
Nondeterministic finite automata

- Nondeterminism
  - A *nondeterministic finite automaton* has the ability to be in several states at once
  - Transitions from a state on an input symbol can be to any set of states
  - Sequence
    - Start in one start state
    - Accept if any sequence of choices leads to a final state
    - Intuitively: the NFA always "guesses right"

Nondeterminism

- Example: Moves on a chessboard
  - States = squares
  - Inputs = r (move to an adjacent red square) and b (move to an adjacent black square)
  - Start state, final state are in opposite corners





← Accept, since final state reached

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

57

Nondeterministic finite automata

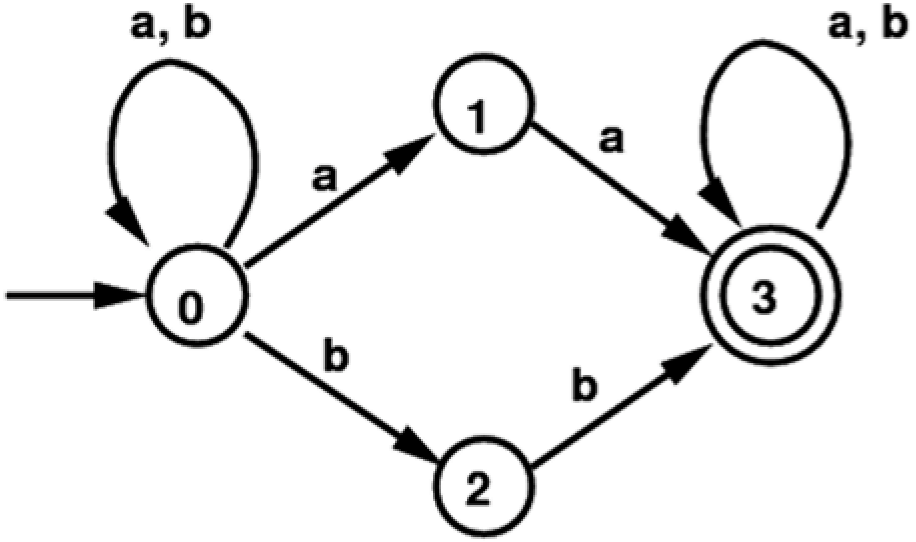
- Formal nondeterministic finite automata (NFA)
  - A finite set of states, typically Q
  - An input alphabet, typically Σ
  - A transition function, typically δ
  - A start state in Q, typically q_0
  - A set of final states $F \subseteq Q$
- Transition function of an NFA
  - $\delta(q, a)$ is a set of states

Nondeterministic finite automata

- Language of an NFA
  - A string w is **accepted** by an NFA if $\delta(q_0, w)$ contains at least one final state
  - That is, **there exists** a sequence of valid transitions from q_0 to a final state given the input w
  - The language of the NFA is the set of strings it accepts

Nondeterministic finite automata

- Example NFA
  - Set of all strings with two consecutive a’s or two consecutive b’s



  - Note that some states have an empty transition on an a or b, and some have multiple transitions on a or b

Nondeterministic finite automata

- Equivalence of DFA's, NFA's
  - DFA → NFA
    - A DFA can be turned into an NFA that accepts the same language
    - If $\delta_D(q, a) = p$, let the NFA have $\delta_N(q, a) = \{p\}$
    - Then the NFA is always in a set containing exactly one state – the state the DFA is in after reading the same input
  - NFA → DFA
    - For any NFA there is a DFA that accepts the same language
    - Proof is the ***subset construction***
    - The number of states of the DFA can be exponential in the number of states of the NFA
    - Thus, NFA's accept **exactly** the regular languages

Nondeterministic finite automata

- Subset construction
  - Given an NFA with states Q, inputs Σ, transition function δ_N, state state q_0, and final states F, construct equivalent DFA with:
    - States 2^Q (Set of subsets of Q)
    - Inputs Σ
    - Start state $\{q_0\}$
    - Final states = all those with a member of F
    - The transition function δ_D is defined by: $\delta_D(\{q_1,\cdots,q_k\}, a)$ is the union over all $i = 1,\cdots,k$ of $\delta_N(q_i, a)$
  - Critical points
    - The DFA states have *names* that are sets of NFA states
    - But as a DFA state, an expression like {p,q} must be read as a single symbol, not as a set

YONSEI UNIVERSITY

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}		
{5}		

Alert: What we're doing here is the *lazy* form of DFA construction, where we only construct a state if we are forced to

YONSEI UNIVERSITY

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

	r	b
→ {1}	{2,4}	{5}
{2,4}	{2,4,6,8}	{1,3,5,7}
{5}		
{2,4,6,8}		
{1,3,5,7}		

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}		
	{1,3,5,7}		
*	{1,3,7,9}		

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}		
*	{1,3,7,9}		
*	{1,3,5,7,9}		

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}		
*	{1,3,5,7,9}		

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}	{2,4,6,8}	{5}
*	{1,3,5,7,9}		

Nondeterministic finite automata

- Example: Subset construction

		r	b
→	1	2,4	5
	2	4,6	1,3,5
	3	2,6	5
	4	2,8	1,5,7
	5	2,4,6,8	1,3,7,9
	6	2,8	3,5,9
	7	4,8	5
	8	4,6	5,7,9
*	9	6,8	5

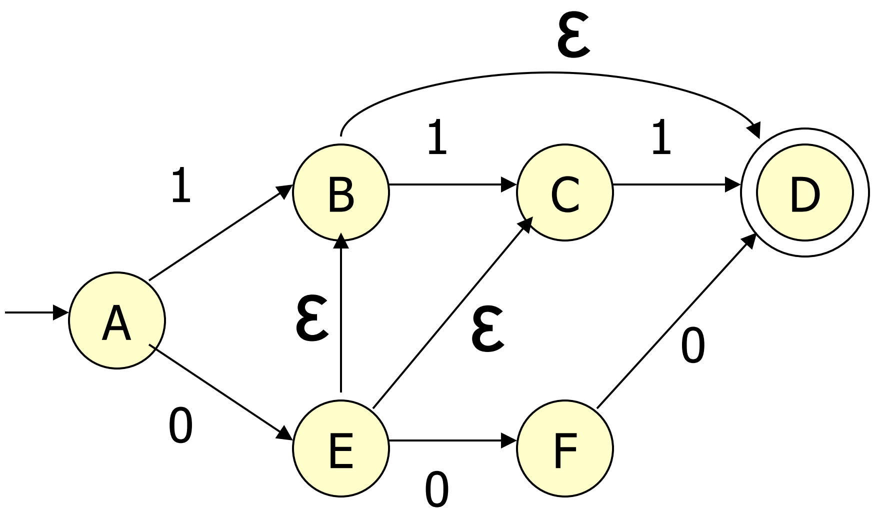
		r	b
→	{1}	{2,4}	{5}
	{2,4}	{2,4,6,8}	{1,3,5,7}
	{5}	{2,4,6,8}	{1,3,7,9}
	{2,4,6,8}	{2,4,6,8}	{1,3,5,7,9}
	{1,3,5,7}	{2,4,6,8}	{1,3,5,7,9}
*	{1,3,7,9}	{2,4,6,8}	{5}
*	{1,3,5,7,9}	{2,4,6,8}	{1,3,5,7,9}

ε–NFA

- NFA’s with ε–transitions
  - We can allow state-to-state transitions on ε input.
  - These transitions are done spontaneously, without looking at the input string.
  - A convenience at times, but still only regular languages are accepted.

ε–NFA

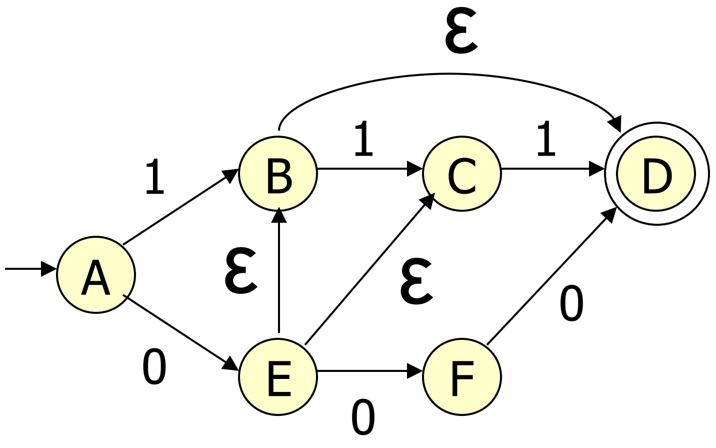
- Example



		0	1	ε
→	A	{E}	{B}	∅
	B	∅	{C}	{D}
	C	∅	{D}	∅
*	D	∅	∅	∅
	E	{F}	∅	{B, C}
	F	{D}	∅	∅

ε–NFA

- Closure of states
  - CL(q) = set of states you can reach from state q following only arcs labeled ε
  - Example
    - CL(A) = {A};
    - CL(E) = {B, C, D, E}
  - Closure of a set of states = union of the closure of each state



ε–NFA

- **Equivalence of NFA, ε–NFA**
  - Every NFA is an ε–NFA
    - It just has no transitions on ε
  - Converse requires us to take an ε–NFA and construct an NFA that accepts the same language
  - We do so by combining ε–transitions with the next transition on a real input

ε–NFA

- Equivalence
  - Start with an ε-NFA with states Q, inputs Σ, start state q_0, final states F, and transition function δ_E
  - Construct an "ordinary" NFA with states Q, inputs Σ, start state q_0, final states F', and transition function δ_N
  - Compute $\delta_N(q, a)$ as follows
    - Let S = CL(q)
    - $\delta_N(q, a)$ is the union over all p in S of $\delta_E(p, a)$
  - F' = the set of states q such that CL(q) contains a state of F
  - Intuition: δ_N incorporates ε–transitions before using *a* but not after

YONSEI UNIVERSITY

ε−NFA

- Example: ε−NFA−to−NFA

Interesting closures:
CL(B) = {B,D};
CL(E) = {B,C,D,E}

		0	1	ε
→	A	{E}	{B}	∅
	B	∅	{C}	{D}
	C	∅	{D}	∅
*	D	∅	∅	∅
	E	{F}	∅	{B, C}
	F	{D}	∅	∅

ε-NFA

		0	1
→	A	{E}	{B}
*	B	∅	{C}
	C	∅	{D}
*	D	∅	∅
*	E	{F}	{C, D}
	F	{D}	∅

Since closures of B and E include final state D.

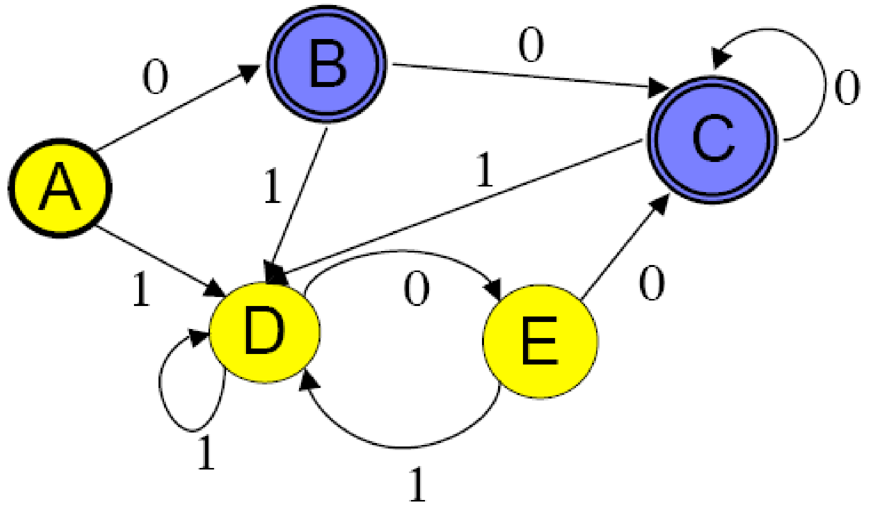
Since closure of E includes B and C; which have transitions on 1 to C and D

Minimization of DFA – supplementary page

- Minimization of DFA
  - DFA is efficient in terms of time (execution time), but inefficient interms of space (number of states)
  - For any regular language **L**, there exists a **unique minimized** DFA **M**.
  - Step 1: partition states into 2 groups: accepting and non-accepting
  - Step 2: in each group, find a sub-group of states having property P
    - P: The states have transitions on each symbol (in the alphabet) to the *same* group
  - Step 3: if a sub-group does not obey P split up the group into a separate group
    - Go back to step 2. If no further sub-groups emerge then continue to step 4
  - Step 4: each group becomes a state in the minimized DFA
    - Transitions to individual states are mapped to a single state representing the group of states

Minimization of DFA – supplementary page

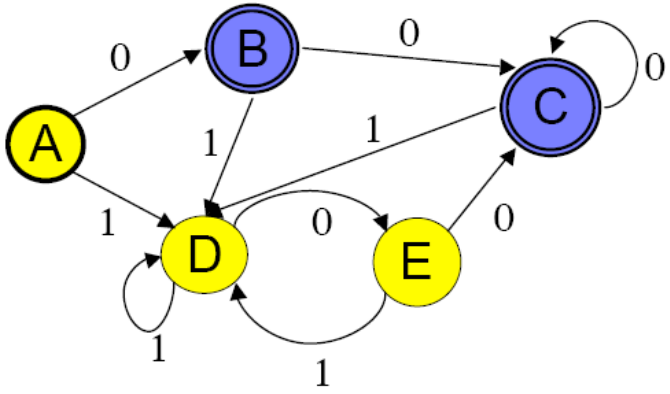
- Example
  - Step 1: partition states into 2 groups: accepting and non-accepting



Minimization of DFA – supplementary page

- Example
  - Step 2: in each group, find a sub-group of states having property P
    - P: The states have transitions on each symbol (in the alphabet) to the *same* group

A, 0: blue
A, 1: yellow
E, 0: blue
E, 1: yellow
D, 0: yellow
D, 1: yellow

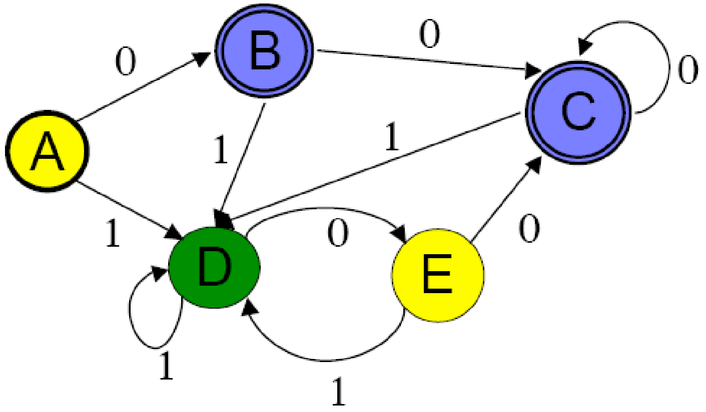


B, 0: blue
B, 1: yellow
C, 0: blue
C, 1: yellow

Minimization of DFA – supplementary page

- Example
  - Step 3: if a sub-group does not obey P split up the group into a separate group
    - Go back to step 2. If no further sub-groups emerge then continue to step 4

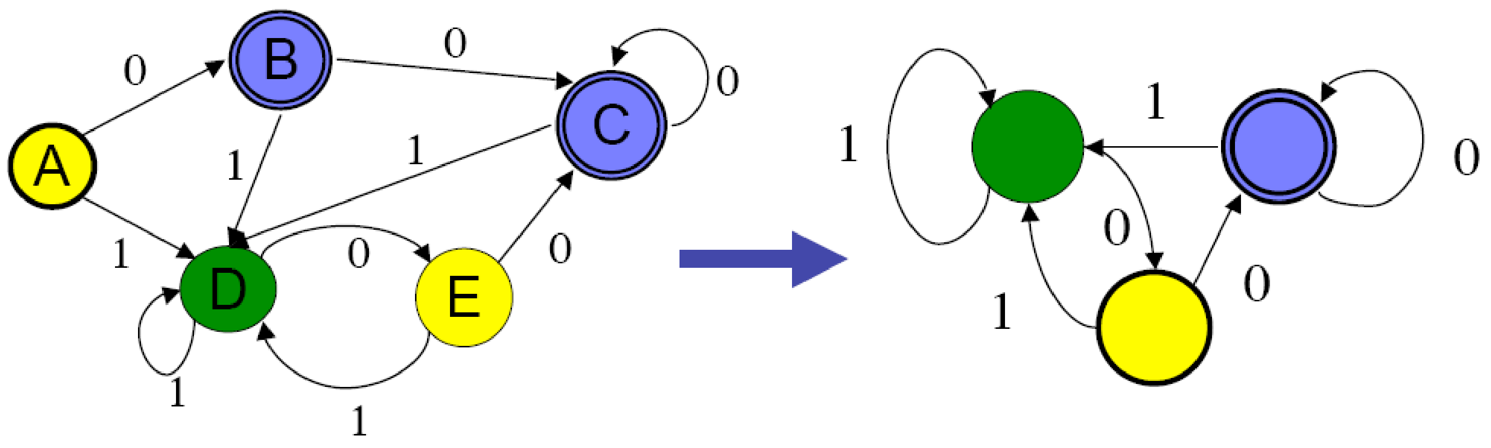
A, 0: blue
A, 1: green
E, 0: blue
E, 1: green
D, 0: yellow
D, 1: green



B, 0: blue
B, 1: green
C, 0: blue
C, 1: green

Minimization of DFA – supplementary page

- Example
  - Step 4: each group becomes a state in the minimized DFA
    - Transitions to individual states are mapped to a single state representing the group of states

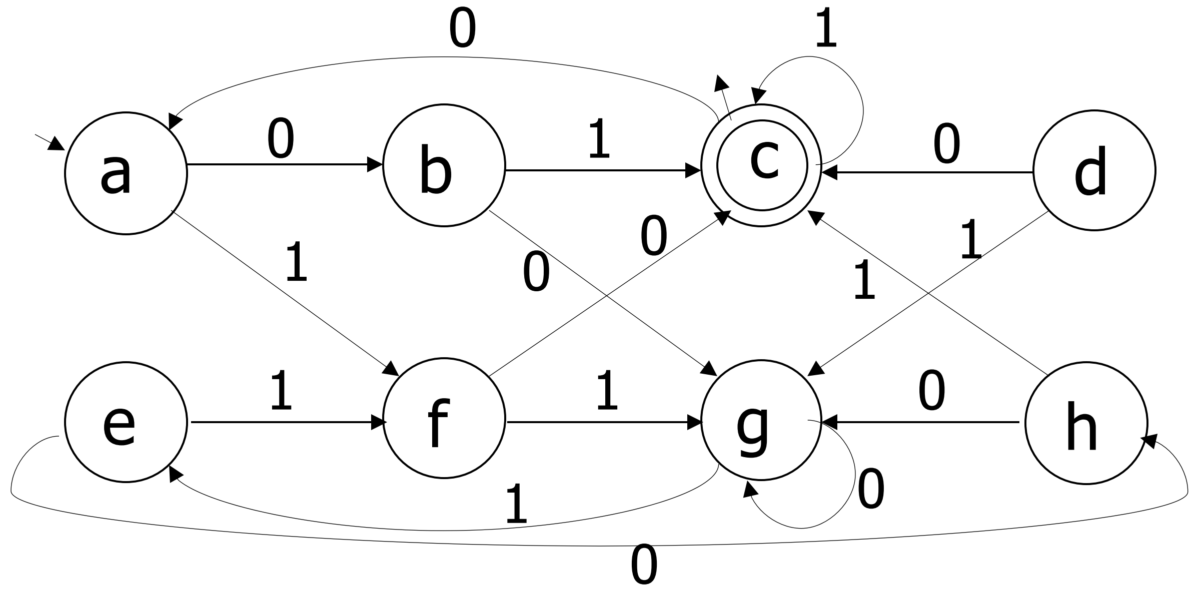


Minimization of DFA

- **Formal definition of equivalent and distinguishable states**
  - Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA, and $\{p, q\} \in Q$.
  - We define **$p \equiv q$** as :
    - For **any** string $w \in \Sigma^*$, $\delta^*(p, w) \in F$ **iff** $\delta^*(q, w) \in F$
    - If $p \equiv q$, we say that p and q are **equivalent**.
  - We define **$p \not\equiv q$** as :
    - There exists **some** string w such that $\delta^*(p, w) \in F$ and $\delta^*(q, w) \notin F$, or vice versa.
    - If $p \not\equiv q$, we say that p and q are **distinguishable**.

Minimization of DFA

- Example



a $\not\equiv$ g? $\delta^*(a, 01) = c \in F$, $\delta^*(g, 01) = e \notin F$

Therefore a $\not\equiv$ g

c $\not\equiv$ g $\delta^*(c, \varepsilon) = c \in F$, $\delta^*(g, \varepsilon) = g \notin F$

Therefore, c $\not\equiv$ g

a $\equiv$ e? yes

w = 01 : $\delta^*(a, 01) = c \in F$, $\delta^*(e, 01) = c \in F$

- w = 0 : $\delta^*(a, 0) = b \notin F$, $\delta^*(e, 0) = h \notin F$
- w = 00110 : $\delta^*(a, 00110) = c \in F = \delta^*(e, 00110)$
- w = 1100110 : ??

. . . .

Minimization of DFA

- Example
  - **Basis** : Find all pairs (p, q) where p ∈ F and q ∉ F ; Then, mark them by √ by as follows:

	a	b	c	d	e	f	g
b							
c	√	√					
d			√				
e			√				
f			√				
g			√				
h			√				

Mark √ in {(a, c), (b, c), (d, c), (e, c), (f, c), (g, c), (h, c)}

Minimization of DFA

- Example
  - Induction

	a	b	c	d	e	f	g
b	√						
c	√	√					
d	√	√	√				
e		√	√	√			
f	√	√	√		√		
g		√	√	√		√	
h	√		√	√	√	√	√

	0	1
(a, b)		(f, c): √
(a, d)	(b, c): √	
(a, f)	(b, c): √	
(a, h)	(b, c): √	
(b, c)		(c, a): √
(b, d)	(g, c): √	
(b, f)	(g, c): √	

Minimization of DFA

- Example
  - Induction

	a	b	c	d	e	f	g
b	√						
c	√	√					
d	√	√	√				
e		√	√	√			
f	√	√	√		√		
g	√	√	√	√	√	√	
h	√		√	√	√	√	√

	0	1
(a, e)	(b, h)	(f, f)
(a, g)	(b, g)	(f, e): √
(b, h)	(g, g)	(c, c)
(d, f)	(c, c)	(g, g)
(e, g)	(h, g): √	(f, e)

YONSEI UNIVERSITY

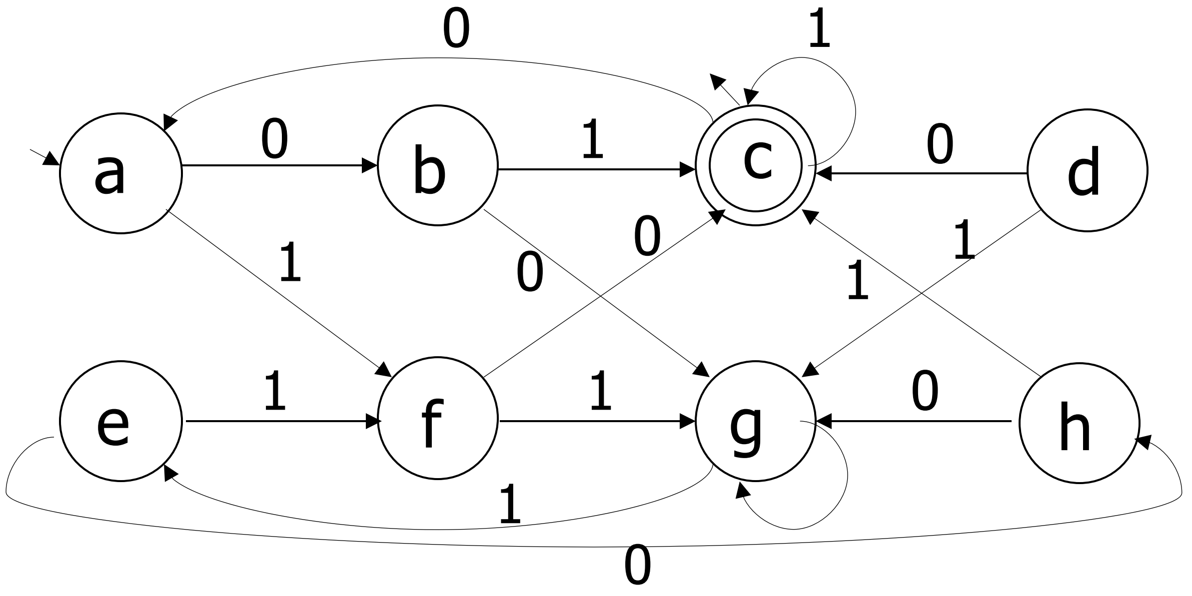
Minimization of DFA

- Example
  - If p and q are <u>not</u> distinguished by this, then p ≡ q

	a	b	c	d	e	f	g
b	√						
c	√	√					
d	√	√	√				
e		√	√	√			
f	√	√	√		√		
g	√	√	√	√	√	√	
h	√		√	√	√	√	√

Equivalent states:
a ≡ e,
b ≡ h,
d ≡ f

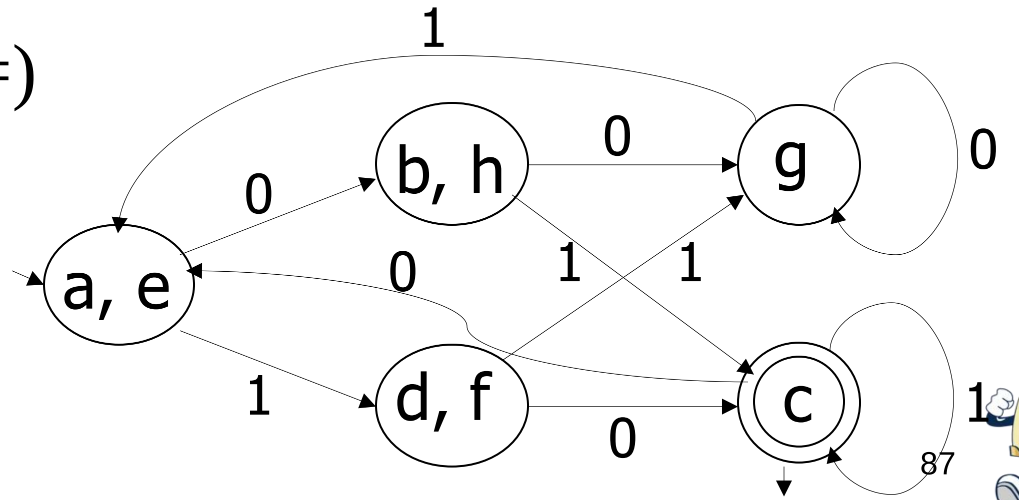
Minimization of DFA



Minimized

Equivalent states:
a ≡ e,
b ≡ h,
d ≡ f

(=)



Regular expression

Regular expression (RE)

- Informal definition
  - A sequence of characters that specifies a match pattern in text
- Formal definition
  - Basis
    - 1) $\emptyset$ is RE
    - 2) ε is RE
      - ε is a regular expression denoting language $\{\varepsilon\}$
    - 3) For any symbol $a \in \Sigma$, a is RE
      - $a \in \Sigma$ is a regular expression denoting $\{a\}$

Regular expression (RE)

- Definition
  - Induction – If r and s are regular expressions denoting languages L(r) and M(s) respectively, then
    - 4) r + s is also RE (union – also notated as r | s)
      - r + s is a regular expression denoting $L(r) \cup M(s)$
    - 5) r · s is also RE (concatenation)
      - rs is a regular expression denoting L(r)M(s)
    - 6) r* is also RE (star closure)
      - r^* is a regular expression denoting $L(r)^*$
    - 7) (r) is also RE (parenthesis)
      - (r) is a regular expression denoting $L(r)$
  - It is represented with three operators, union(+), concatenation(·), star closure(*)
    - Excludes (r) for the convenience

Regular expression (RE)

- Example
  - Let $\Sigma = \{a, b\}$
    - $a + b$
    - $(a + b)(a + b)$
    - a^*
    - $(a + b)^*$
    - $a + a^*b$

YONSEI UNIVERSITY

Regular expression (RE)

- Precedence of opeartors

$$* > \cdot > +$$

  - $0 \cdot 1^* + 1 = (0 \cdot (1)^*) + 1$
  - Other examples
    - $(0 + 1)^*00(0+1)^*$
    - $(1 + 10)^*$
    - $(0 + \varepsilon)(1 + 10)^*$
    - $(00)^*(11)^*1 = \{0^{2n}1^{2m+1} \mid n, m > 0\}$
    - $(0 + 1)^*(00 + 1)$
    - $(\text{letter})\ (\text{letter} + \text{digit})^*$
    - $(01)^* + (10)^* + 0(10)^* + 1(01)^*$
    - $(0 + 1)^*(00 + 01 + 10 + 11)^*(0 + 1)^*$

Regular expression (RE)

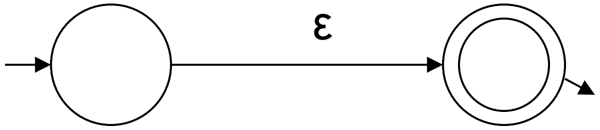
- Arithmetic law
  - $r + s = s + r$ (commutative)
  - $r + (s + t) = (r + s) + t$ (associative)
  - $(r \cdot s) \cdot t = r \cdot (s \cdot t)$ (associative)
  - $r \cdot (s + t) = (r \cdot s) + (r \cdot t)$ (distributive)
  - $(r + \varepsilon)^* = r^*$ (guaranteed in a closure)
  - $r \cdot \varepsilon = r = \varepsilon \cdot r$ (identity element for concatenation)
  - $r \cdot \emptyset = \emptyset = \emptyset \cdot r$
  - $r + r \cdot s^* = r \cdot s^*$
  - $(r^*)^* = r^*$ (* is idempotent)
  - $r \cdot r^* = r^+ = r^* \cdot r$

YONSEI UNIVERSITY

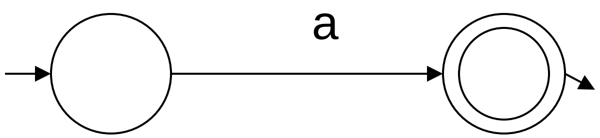
Regular expression (RE)

- Convert R.E. into ε−NFA
  - For every R.E. **R**., we can construct ε−NFA **M** such that **L(R) = L(M)**.
  - (Basis)
    - R = **∅**:

    - R = ε:



    - R = a (∈ Σ):

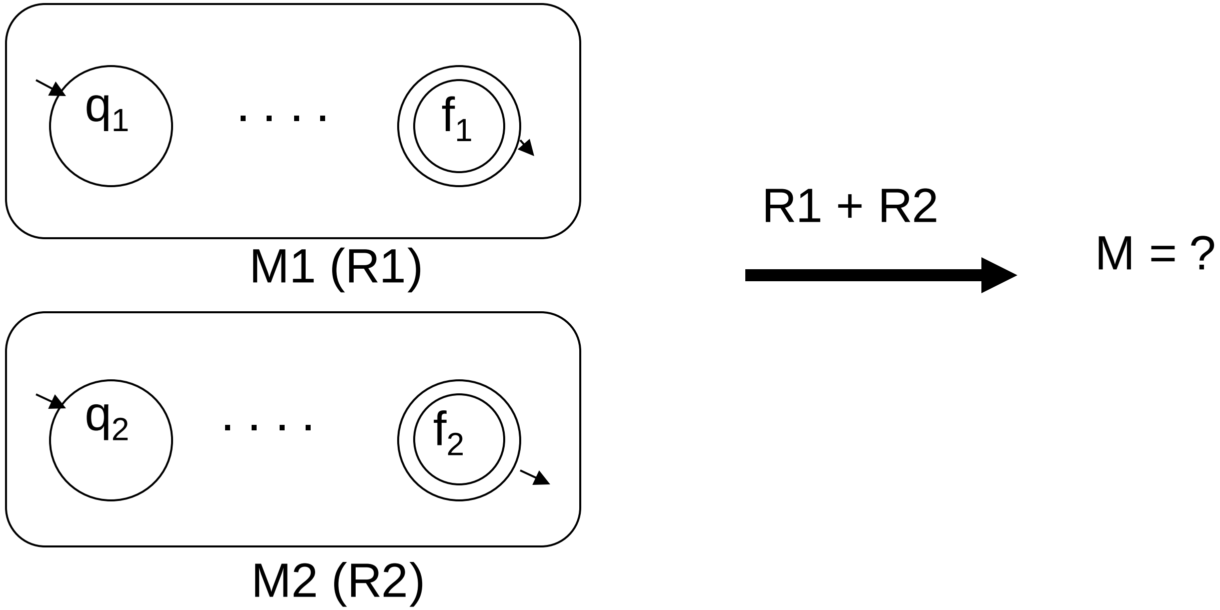


Regular expression (RE)

- Convert R.E. into ε−NFA
  - (Induction Step)
    - Let
      - M1 = (Q_1, Σ_1, δ_1, q_1, {f_1}) be a ε−NFA for R1
      - M2 = (Q_2, Σ_2, δ_2, q_2, {f_2}) be a ε−NFA for R2

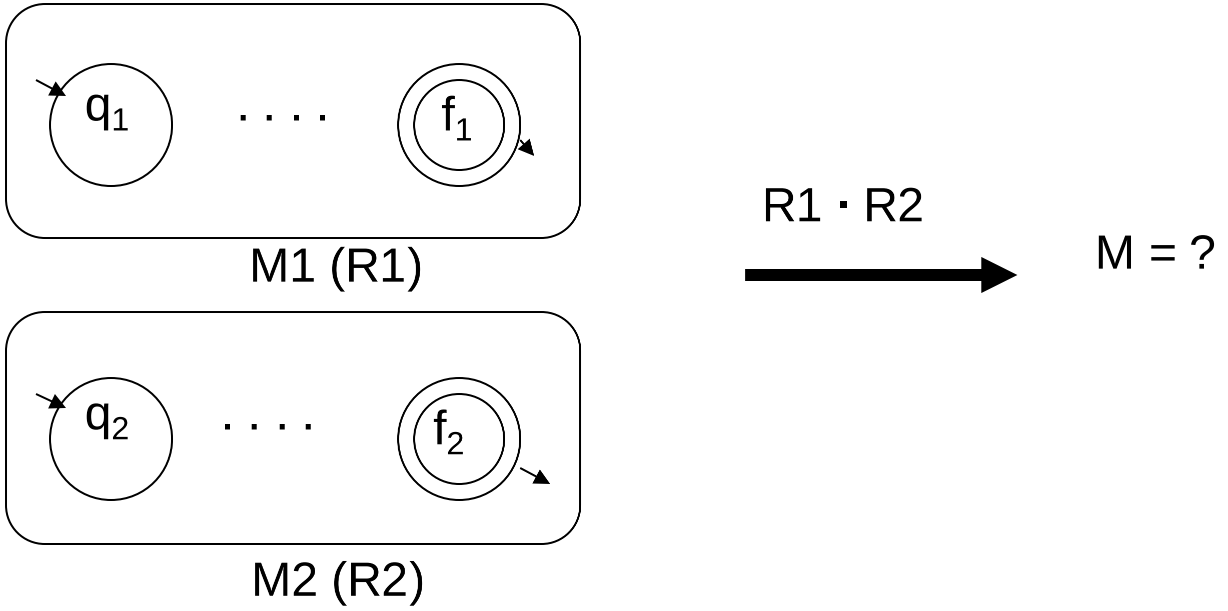
Regular expression (RE)

- Convert R.E. into ε-NFA
  - (Induction Step)
    - Case 1: R1 + R2 :
      - We construct ε-NFA M for R1 + R2 as follows: $M = (Q_1 \cup Q_2 \cup \{q_0, f_0\}, \Sigma_1 \cup \Sigma_2, \delta, q_0, \{f_0\})$



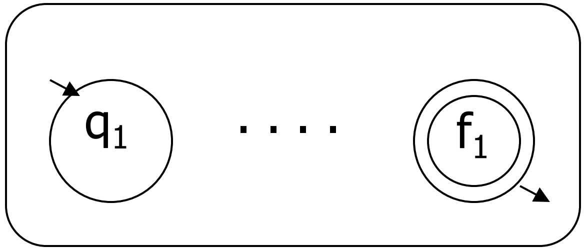
Regular expression (RE)

- Convert R.E. into ε−NFA
  - (Induction Step)
    - Case 2: R1 · R2 :
      - We construct ε−NFA M for R1 · R2 as follows: $M = (Q_1 \cup Q_2, \Sigma_1 \cup \Sigma_2, \delta, q_1, \{f_2\})$



Regular expression (RE)

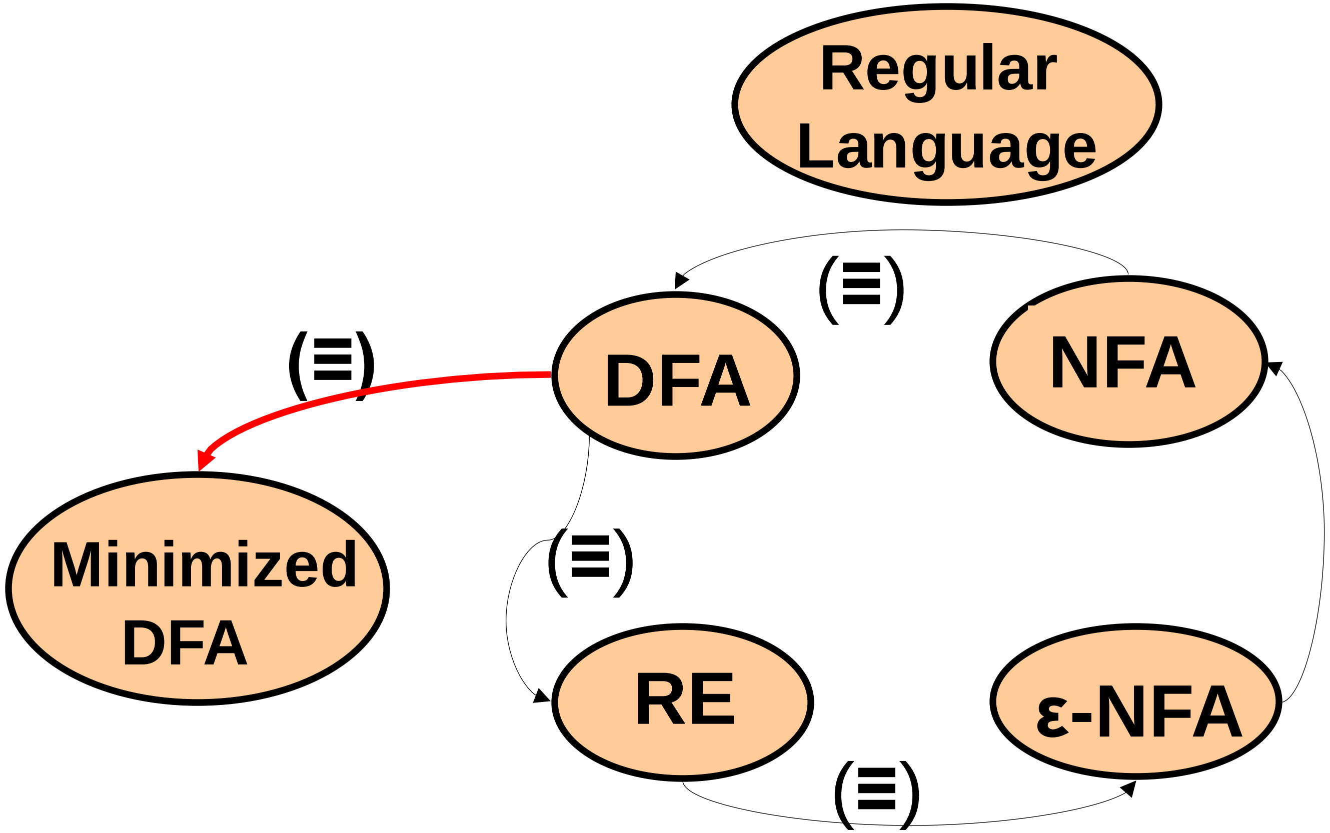
- Convert R.E. into ε-NFA
  - (Induction Step)
    - Case 3: R1* :
      - We construct ε-NFA M for R1* as follows:
        $M = (Q_1 \cup \{q_0, f_0\}, \Sigma_1, \delta, q_0, \{f_0\})$



M1 (R1)

$\xrightarrow{R1^*}$ M = ?

Equility



Questions?