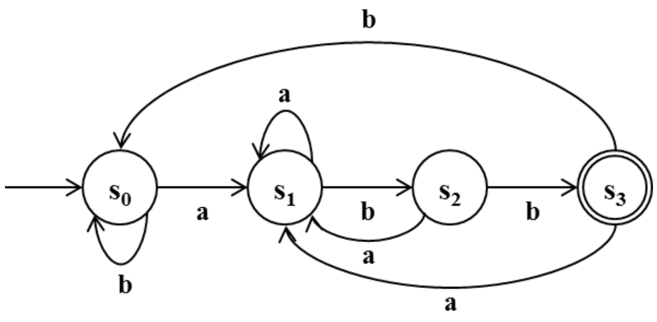
Lexical Analysis



(a|b)*abb

Recap: Basic Structure of a Compiler

- Basic Structure of a Compiler
  1. Lexical Analysis
  2. Syntax Analysis
  3. Semantic Analysis
  4. Optimization
  5. Code Generation

acknowledgements: contains adapted material from the Dragon Book / Alex Aiken

Lexical Analysis Contents

- Lexical Specification
- Finite Automata
- From Regular Expressions to NFA
- From NFA to DFA
- Minimizing the Number of States in a DFA
- Simulating an NFA

acknowledgements: contains adapted material from the Dragon Book / Alex Aiken

Lexical Analysis Contents

Lexical Specification

↓

Regular Expression

↓

NFA

↓

DFA

↓

Table-driven DFA Implementation

Lexical Analysis

- Formally:
  - input: character stream of source program
  - process:
    - split stream into *lexemes* according to the *grammar* of the language
    - build attributed *tokens*
  - output: token stream
    - a token contains the token name (id) and the token attributes

      < token name, attribute values >

Lexical Analysis

- Example:

position = initial + rate * 60

1. lexeme "position" → token <id, 1>
2. lexeme "=" → token <=>
3. lexeme "initial" → token <id, 2>
4. lexeme "+" → token <+>
5. lexeme "rate" → token <id, 3>
6. lexeme "*" → token <*>
7. lexeme "60" → token <number, 60>

<id,1> <=> <id,2> <+> <id,3> <*> <number,60>

Lexical Analysis

- Input stream is not as easy to read:

```
\t\t\t  position =\n\t\t\t\t\tinitial +\trate * 60;
```

  - a stream of characters from a source file, including tabs, newlines, and spaces

```
$ cat example.cpp
#include <iostream>
using namespace std;


template <class T>
class A {
  T *ref;

  public:

  void setref(T *r);
  T*   getref(void);
};

...

  cout << *j << endl;

  return 0;
}
```

```
$ od -c exmple.cpp
0000000   #   i   n   c   l   u   d   e       <   i   o   s   t   r   e
0000020   a   m   >  \n   u   s   i   n   g       n   a   m   e   s   p
0000040   a   c   e       s   t   d   ;  \n  \n  \n   t   e   m   p   l
0000060   a   t   e       <   c   l   a   s   s       T   >  \n   c   l
0000100   a   s   s       A       {  \n           T       *   r   e   f
0000120   ;  \n  \n           p   u   b   l   i   c   :  \n  \n        
0000140   v   o   i   d       s   e   t   r   e   f   (   T       *   r
0000160   )   ;  \n           T   *               g   e   t   r   e   f
0000200   (   v   o   i   d   )   ;  \n   }   ;  \n  \n   t   e   m   p
0000220   l   a   t   e       <   c   l   a   s   s       T   >  \n   v
0000240   o   i   d       A   <   T   >   :   :   s   e   t   r   e   f
0000260   (   T       *   r   )  \n   {  \n           r   e   f       =
0000300       r   ;  \n   }  \n  \n   t   e   m   p   l   a   t   e    
0000320   <   c   l   a   s   s       T   >  \n   T   *       A   <   T
0000340   >   :   :   g   e   t   r   e   f   (   v   o   i   d   )  \n
...
0000560   e   n   d   l   ;  \n  \n           r   e   t   u   r   n    
0000600   0   ;  \n   }  \n  \n  \n
```

Tokens, Patterns, and Lexemes

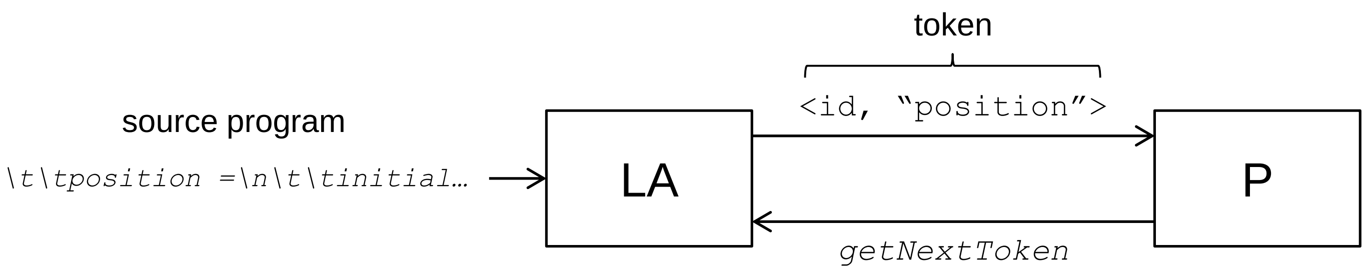
- Token
  - `<token name/id, optional attribute>`
  - often, the optional attribute is the lexeme
- Pattern
  - rule defining the possible lexemes of a token
- Lexeme
  - sequence of characters that matches the pattern of a token
- Example: C identifiers
  - pattern: “an identifier starts with a letter or an underscore, then continues with letters, numbers, or underscores.”
  - lexemes: “i”, “k”, “_tmp”, “_private1”, “__tmp_value”, “_1”, “_”, “if”, “while”, …
  - tokens: `<id, “i”>, <id, “k”>, <id, “_tmp”>,` …

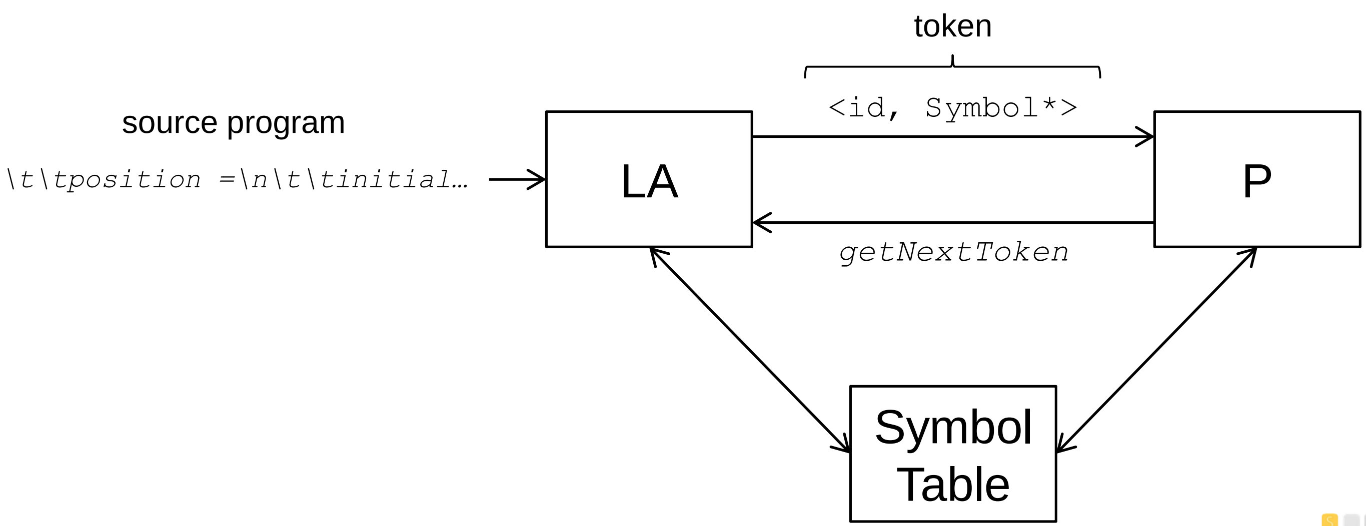
Typical Token Classes

- Token classes
  - identifiers
  - keywords
  - integers
  - operators
  - separators
  - whitespace

Role of Lexical Analysis

- Tokenize the substrings from the input according to role
- Deliver tokens to the parser





Basic Algorithm for Lexical Analysis

- getNextToken()

```
given: patterns that define the input language

lexeme = '';

while (input stream not empty) do begin
  append next character in input stream to lexeme;

  if (lexeme matches a pattern) then begin
    return token<token id, lexeme>
  end
end
```

Lookahead in Lexical Analysis

- Identifiers vs. Keywords

```
elsewhere = iffiness * 60;
```

- =, <, >, <=, >=, ==, <<, >>

```
if (a == b)
```

```
Template<int> a;
Template<Template<int>> b;
```

Lookahead in Lexical Analysis

- Input string is read left-to-right, tokens are recognized as we go

- A **lookahead** is often required to decide whether a token has ended

```
elsewhere
```

seen

lookahead

start of lexeme

```
if (a == b){
```

```
Template<Template<int>> a;
```

- is one character enough?

Lexical Analysis Caveats

- FORTRAN I

  whitespace is insignificant

  these are all the same:
  `VARIABLE, VA RI AB LE, VARI A          BLE`

- Loop or variable?

  `DO 5 I = 1.25`

  `DO 5 I = 1,25`

  lookahead

- is the lookahead bounded?

Lexical Analysis Caveats

- PL/1

  keywords are not reserved

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

- Unbounded lookahead

```
DECLARE (ARG1, ARG2, …, ARGN) =

DECLARE (ARG1, ARG2, …, ARGN) ;
```

lookahead

Pattern Specification for Lexical Analysis

- Lexical structure of input is specified by the set of patterns that define the lexemes of the language

- Regular expressions well-suited for this purpose
  - easy to specify patterns
  - build an NFA, then a DFA from a regexp
  - format for specification for automatic LA generators (Lex/Flex/…)

Regular Expressions

Nomenclature for Parts of a String

- For a string s = compiler,
  - prefix
    resulting string obtained by removing 0 or more symbols from the end of s
    - c, com, compiler, ε
  - suffix
    resulting string obtained by removing 0 or more symbols from the start of s
    - r, ler, compiler, ε
  - substring
    resulting string obtained by removing any prefix and suffix from s
    - compiler, ompiler, omp, ile, ε
  - subsequence
    formed by deleting zero or more not necessarily consecutive symbols of s
    - cmlr, cpe, compiler, ε
  - *proper* prefix, suffix substring
    resulting string is neither s nor ε

Alphabet, String, and Language

- Alphabet Σ
  - any finite set of symbols
  - examples
    - { 0, 1 }, { a, b, c, d }
    - { ”<”, ”>”, ”+”, ”-”, ”.”, ”,”, ”[”, ”]” }
    - { ■, □, ▲, ▸, ▼, ♥, ♠, ☻, ☺, ♪, ♫ }

- String s
  - aka “word”, “sentence”
  - a finite sequence of symbols drawn from the alphabet
  - length of string $|s|$ = number of symbol occurrences in s
  - empty string ε, with $|\ \varepsilon\ | = 0$
  - concatenation for strings x, y, denoted xy = appending y to x
  - exponentiation of a string s: $s^0 = \varepsilon$, $s^1 = s$, $s^2 = ss$, $s^3 = sss$, …

Alphabet, String, and Language

- Language
  - any countable set of strings over some fixed alphabet Σ
  - examples
    - set of all strings containing exactly 4 symbols from the alphabet
    - set of all syntactically well-formed C programs
    - set of all grammatically correct English sentences
    - $\emptyset$
    - $\{ \varepsilon \}$
  - note: there is no meaning ascribed to the strings in the language

Operations on Languages

- Four most important operations on languages

Operation	Notation	Definition
Union	$L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation	LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure	L^*	$L^* = \cup_{i=0,\infty} L^i$
Positive closure	L^+	$L^+ = \cup_{i=1,\infty} L^i$

Operations on Languages

- Example

  For

  $L = \{ A, B, \ldots, Z, a, b, \ldots, z \}$
  $D = \{ 0, 1, \ldots, 9 \}$

  - $L \cup D = \{ A, B, \ldots, Z, a, b, \ldots, z, 0, 1, \ldots, 9 \}$
  - $LD = \{ A0, A1, \ldots, A9, B0, B1, \ldots, B9, \ldots, Z0, Z1, \ldots, Z9 \}$
  - L^* = set of all strings of letters (including ε)
  - L^+ = set of all strings of letters (excluding ε)
  - $L(L \cup D)^*$ = set of all strings of letters and digits beginning with a letter

Regular Expressions

- Built recursively over some alphabet Σ
  - basis
    - ε is a regexp, $L(\varepsilon) = \{\varepsilon\}$
    - if $a \in \Sigma$, then $\mathbf{a}$ is a regexp, and $L(\mathbf{a}) = \{\mathrm{a}\}$
  - induction
    for regexps r and s, denoting the languages $L(r)$ and $L(s)$
    - $(r)|(s)$ is a regexp denoting $L(r) \cup L(s)$
    - $(r)(s)$ is a regexp denoting $L(r)L(s)$
    - $(r)^*$ is a regexp denoting $(L(r))^*$
    - (r) is a regexp denoting $L(r)$

- Operator precedence
  - * > concatenation > |
  - all left-associative

Regular Expressions

- Examples

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

  - **a**|**b** denotes {a, b}
  - (**a**|**b**)(**a**|**b**) denotes {aa, ab, ba, bb}
  - **a*** denotes {ε, a, aa, aaa, aaaa, … }
  - (**a**|**b**)* denotes {ε , a, b, aa, ab, ba, bb, aaa, aab, …}
  - **a**|**a*b** denotes {a , b, ab, aab, aaab, …}

Algebraic Laws for Regular Expressions

<table>
<tr><th>Law (Axiom)</th><th>Description</th></tr>
<tr><td>r | s = s | r</td><td>| is commutative</td></tr>
<tr><td>r (s | t) = (r | s) | t</td><td>| is associative</td></tr>
<tr><td>r(st) = (rs)t</td><td>concatenation is associative</td></tr>
<tr><td>r(s|t) = rs|rt
(s|t)r = sr|tr</td><td>concatenation distributes over |</td></tr>
<tr><td>εr = rε = r</td><td>ε is the identity element for concatenation</td></tr>
<tr><td>r* = (r|ε)*</td><td>relation between * and ε:
ε is guaranteed in a closure</td></tr>
<tr><td>r** = r*</td><td>* is idempotent</td></tr>
</table>

Regular Definitions

- For notational convenience

- Given an alphabet Σ, a regular definition is a sequence of definitions of the following form

$$\begin{aligned} &d_1 \rightarrow r_1 \\ &d_2 \rightarrow r_2 \\ &\ldots \\ &d_n \rightarrow r_n \end{aligned}$$

  where

  - each d_i is a new symbol (not in Σ, and $d_i \neq d_j$ for $i \neq j$)
  - each r_i is a regular expression over the alphabet $\Sigma \cup \{ d_1, d_2, d_3, \ldots d_{i-1} \}$

Regular Definitions

- Example

  - C identifiers

    letter_ → A | B | … | Z | a | b | … | z | _
    digit → 0 | 1 | 2 |… | 9
    ident → letter_ (letter_ | digit)*

  - Unsigned integer or floating point numbers

    digit → 0 | 1 | 2 |… | 9
    digits → digit digit*
    optFract → . digits | ε
    optExp → (E (+ | - | ε) digits) | ε
    number → digits optFract optExp

Extensions of Regular Expressions

- Instance counting
  - one or more
    $r^+ = rr^*$
  - zero or one
    $r? = r \mid \varepsilon$
  - n instances
    $r^n = \underbrace{r \ldots r}_{\text{n times}}$
- Character classes
  - replace a series of | with [], and for logical sentences use
    $a_1|a_2|\ldots|a_n = [a_1a_2\ldots a_n]$
    [abcdefghijklmnopqrstuvwxyz] = [a-z]

Regular Definitions

- Example using the extensions
  - C identifiers

letter_ → [A-Za-z_]
digit → [0-9]
ident → letter_ (letter_ | digit)*

  - Unsigned integer or floating point numbers

digits → $[0\text{-}9]^+$
number → digits (. digits)? (E [+-]? digits)?

Recap: Languages

- Language
  - any countable set of strings over some fixed alphabet Σ
  - important: there is no meaning ascribed to the strings in the language
  - operations on languages

Operation	Notation	Definition
Union	$L \cup M$	$L \cup M = \{s \mid s \in L \text{ or } s \in M\}$
Concatenation	LM	$LM = \{st \mid s \in L \text{ and } t \in M\}$
Kleene closure	L^*	$L^* = \cup_{i=0,\infty} L^i$
Positive closure	L^+	$L^+ = \cup_{i=1,\infty} L^i$

Recap: Regular Expressions

- Built recursively over some alphabet Σ
  - basis
    - ε is a regexp, $L(\varepsilon) = \{\varepsilon\}$
    - if $a \in \Sigma$, then $\mathbf{a}$ is a regexp, and $L(\mathbf{a}) = \{\mathrm{a}\}$
  - induction
    for regexps r and s, denoting the languages $L(r)$ and $L(s)$
    - $(r)|(s)$ is a regexp denoting $L(r) \cup L(s)$
    - $(r)(s)$ is a regexp denoting $L(r)L(s)$
    - $(r)^*$ is a regexp denoting $(L(r))^*$
    - (r) is a regexp denoting $L(r)$

- Operator precedence
  - * > concatenation > |
  - all left-associative

Recap: Regular Expressions

- **Regular Definitions**
  - definitions of the form $d_i \rightarrow r_i$ where
    - each d_i is a new symbol (not in Σ, and $d_i \neq d_j$ for $i \neq j$)
    - each r_i is a regular expression over the alphabet $\Sigma \cup \{ d_1, d_2, d_3, \ldots d_{i-1} \}$

- **Extensions of Regular Expressions**
  - counting
    - one or more: $r^+ = rr^*$
    - zero or one: $r? = r \mid \varepsilon$
    - n instances: $r^n = r\ldots r$
  - character classes:
    - range: $a_1|a_2|\ldots|a_n = [a_1a_2\ldots a_n]$, [abcdefghijklmnopqrstuvwxyz] = [a-z]
    - excluded range: [^a-c] = complement of [a-c]

Recap: RegExp and Languages

- L(α) is function that gives meaning to α

- For regular expressions

  L: exp $\rightarrow$ set of strings over Σ

- Separation of syntax (notation) and semantics (meaning)
  - treat syntax as a separate issue
  - syntax $\leftrightarrow$ semantics is not 1:1
    - many-to-one, but never one-to-many

Lexical Specification

Specifying Syntax through Regular Expressions

- Two Goals of Lexical Analysis

  given a string s (the program source) and a regular expression R

  - determine $s \in L(R)$

    and

  - tokenize s into substrings

Lexical Specification

1. Specify a regular expression for each syntactic category
   - **Keyword** = ‘if’ | ‘else’ | ‘then’ | …
   - **Number** = digit+ optFract optExp
   - **Identifier** = letter (letter | digit)*
   - **Op** = ‘+’ | ‘-’ | ‘*’ | ‘/’ | …
   - **RelOp** = ‘<=‘ | ‘==‘ | ‘>=‘ | …
   - **LPar** = ‘(‘
   - **RPar** = ‘)’
   - **Whitespace** = [\n\t]
   - …

Lexical Specification

2. Form the lexical specification of the language

   R = Keyword | Number | Identifier | Op | RelOp | LPar | RPar | Whitespace | …
   = R_1 | R_2 | …

   - simply the union of all regular expressions

Lexical Analysis

CSE 컴퓨터공학부
Department of Computer Science & Engineering

Algorithm for Lexical Analysis

- Given: R matching all lexemes for all tokens, i.e., our language
  Input: $\mathrm{x}_1\mathrm{x}_2\ldots\mathrm{x}_\mathrm{n}$

1. For $1 \leq \mathrm{i} \leq \mathrm{n}$ check

   $$\mathrm{x}_1\mathrm{x}_2\ldots\mathrm{x}_\mathrm{i} \in L(R)$$

2. Since R is the union of all R_j

   $$\exists\, j\colon \mathrm{x}_1\mathrm{x}_2\ldots\mathrm{x}_\mathrm{i} \in L(R_j)$$

3. Delete $\mathrm{x}_1\mathrm{x}_2\ldots\mathrm{x}_\mathrm{i}$ from input and go to 1

Algorithm for Lexical Analysis

- Ambiguities
  - how much input should be consumed?

$$x_1 \ldots x_i \in L(R)$$
$$x_1 \ldots x_j \in L(R)$$
$$i \neq j$$

- examples
  - `tmp` vs `tmp1`
  - `else` vs `elsewhere`
  - `=` vs `==`

Rule: always pick the longer one ("maximal munch")

Algorithm for Lexical Analysis

- Ambiguities
  - which token should be generated?

$$x_1 \ldots x_i \in L(R)$$

$$x_1 \ldots x_i \in L(R_k)$$
$$x_1 \ldots x_i \in L(R_l)$$
$$k \neq l$$

- example

`else` vs `else` (keyword vs identifier)

Rule: priority ordering amongst the R_k; and choose the one listed first

Algorithm for Lexical Analysis

- Ambiguities
  - what if no rule matches?

$$x_1 \ldots x_i \notin L(R)$$

Several possibilities:

- print error directly in lexical analyzer
- add a rule to the end of the specification that catches any input *not* in the specification

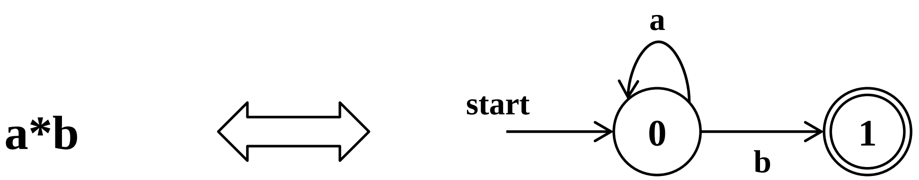
Error = "all strings not in the specification"

implemented as 'wildcard' & put last
→ problem here: recovery

Finite Automata

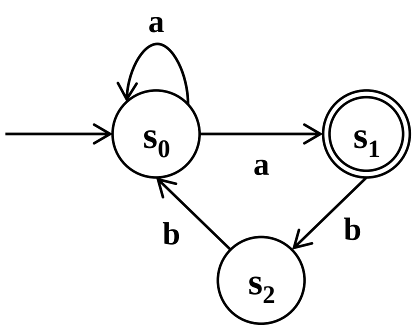
Regular Expressions and Finite Automata

- Specification = Regular Expression (RE)
- Implementation = Finite Automata (FA)
- REs and FA are closely related
  - both describe regular languages
  - convert from one into the other



Finite Automata

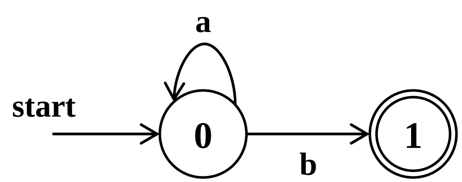
- A finite automaton consists of
  - a set of input symbols Σ
    - does not include the empty string ε
  - a finite set of states S
  - a start state $s_0 \in S$
  - a set of final (or accepting) states $F \subseteq S$
  - a transition function $f : S \times (\Sigma \cup \varepsilon) \rightarrow S$



state	a	b	ε
0	{0, 1}	∅	∅
1	∅	2	∅
2	∅	0	∅

Finite Automata

- Transitions in FA
  - the transition function $f : S \times (\Sigma \cup \varepsilon) \rightarrow S$ defines where to go from the current state and the next input symbol
  - example



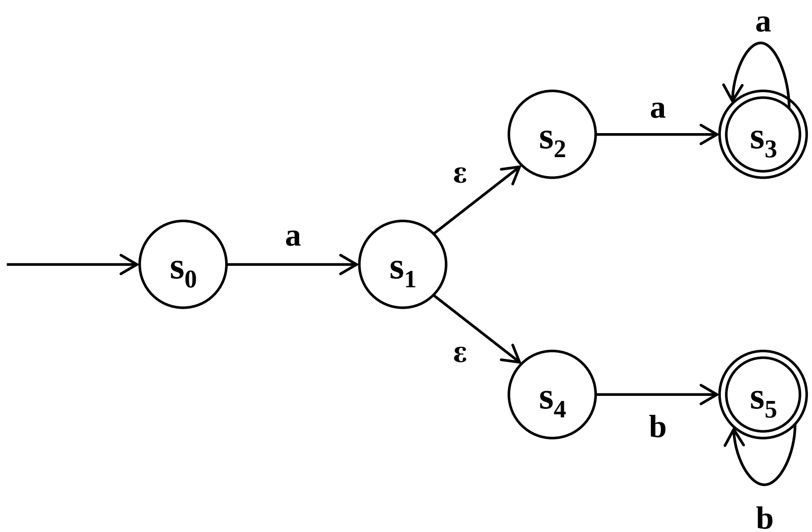
    - state 0 on input a → state 0
    - state 0 on input b → state 1

- Acceptance
  - if there exists *some* path that leaves the FA is in an accepting state at the end of the input

Nondeterministic Finite Automata (NFA)

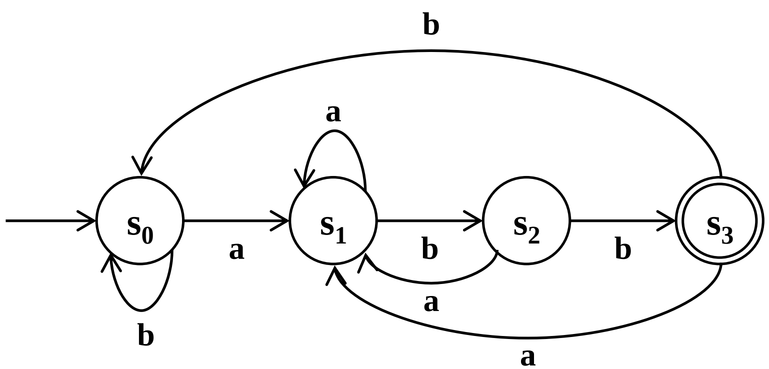
- Can have several transitions (edges) out of a state on the same input symbol

- Can have ε transitions
  - move from one state to the other without consuming any input



Deterministic Finite Automata (DFA)

- Can have exactly one transition (edge) out of a state on the same input symbol
- No ε transitions
  - must consume an input symbol on every transition



(a|b)*abb

state	a	b
0	1	0
1	1	2
2	1	3
3	1	0

Simulating a DFA

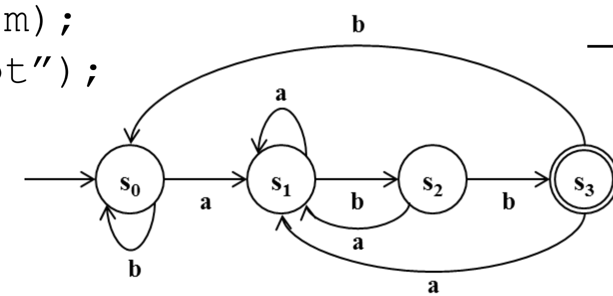
- Simulating a DFA is straightforward

```
int DFA(char *input, int s0, int (*move)(int, char))
{
  int s = s0;
  char c = *input++;
  while (c != '\0') {
    s = move(s, c);
    c = *input++;
  }
  return s
}
```

```
int mtab[S][Σ];

int m(int s, char c)
{
  return mtab[s][c];
}
```

usage:

```
int f = DFA("aababb", 0, m);
if (f == 3) printf("accept");
else printf("nope");
```

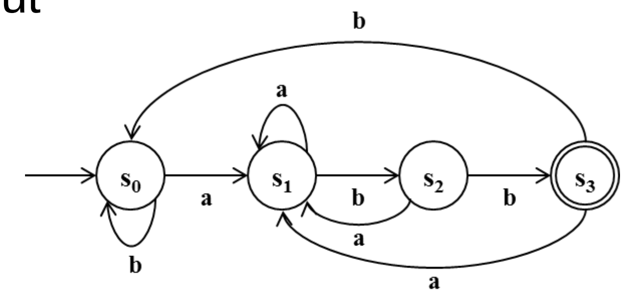


(a|b)*abb

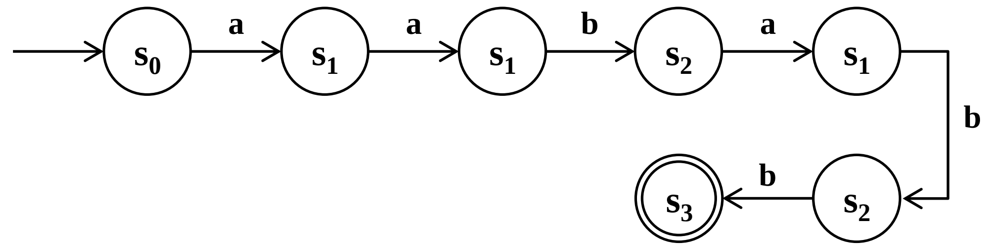
state	a	b
0	1	0
1	1	2
2	1	3
3	1	0

NFA vs DFA

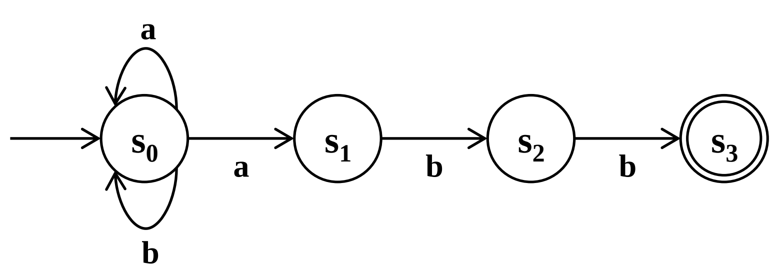
- A DFA takes exactly one, well-defined path through the state graph for any given input



aababb

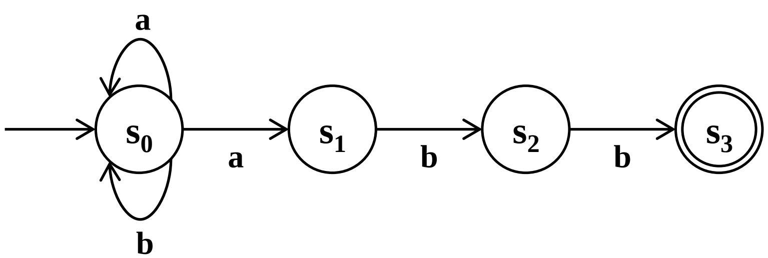


- An NFA can "choose"
  - an NFA can conceptually be in any number of states
    → simulate all possible paths through the NFA in parallel



NFA vs DFA

- NFA Example
  NFA accepting the same regular expression as the DFA before



(a|b)*abb

input: **aababb**

input	state set
	{ 0 }
a	{0, 1}
a	{0, 1}
b	{0, 2}
a	{0, 1}
b	{0, 2}
b	{0, **3**}

An NFA is typically in a *set* of states, not one *single* state.

NFA vs DFA

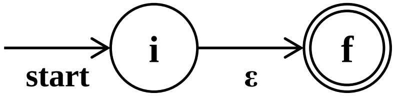
- NFA and DFA both recognize the set of same languages (regular languages)
  - for each NFA recognizing a certain language, a corresponding DFA exists
  - and vice-versa
- NFA and DFA differ in construction complexity and recognizing speed
  - space/time tradeoff
  - construction:
    - NFAs are more compact
  - execution:
    - DFAs are faster

From Regular Expressions to NFA

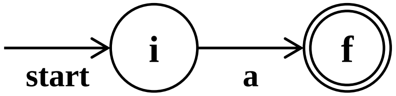
CSE 컴퓨터공학부 Department of Computer Science & Engineering

Thompson's Construction Algorithm

- Inductive construction of an NFA from a regular expression
  - input: regular expression
  - output: NFA
  - method: break down RE into base components and compose according to the following basic rules
  - base:
    - for ε:

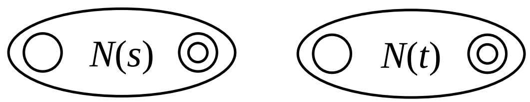


    - for input $\mathbf{a} \in \Sigma$:

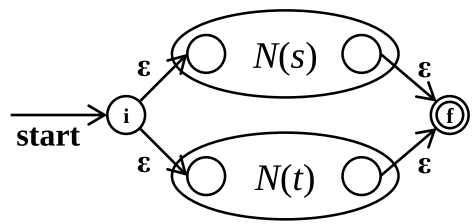


Thompson's Construction Algorithm

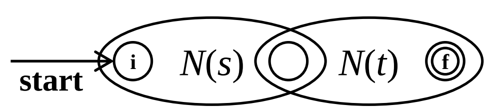
- Inductive construction of an NFA from a regular expression (cont'd)
  - given: the NFA $N(s)$, $N(t)$ for regular expressions s, t



  - composition:
    - r = s|t:

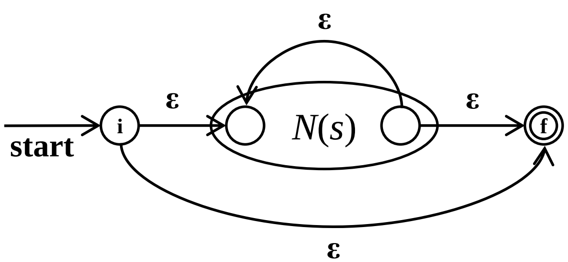


    - r = st:

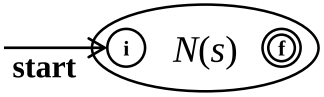


Thompson's Construction Algorithm

- Inductive construction of an NFA from a regular expression (cont'd)
  - composition:
    - r = s*:



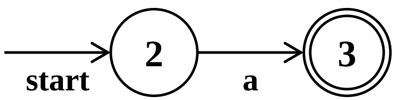
    - r = (s):



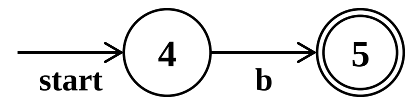
Thompson’s Construction Algorithm

- Example: **(a|b)*abb**

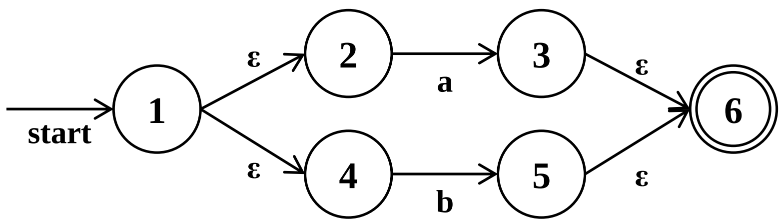
a



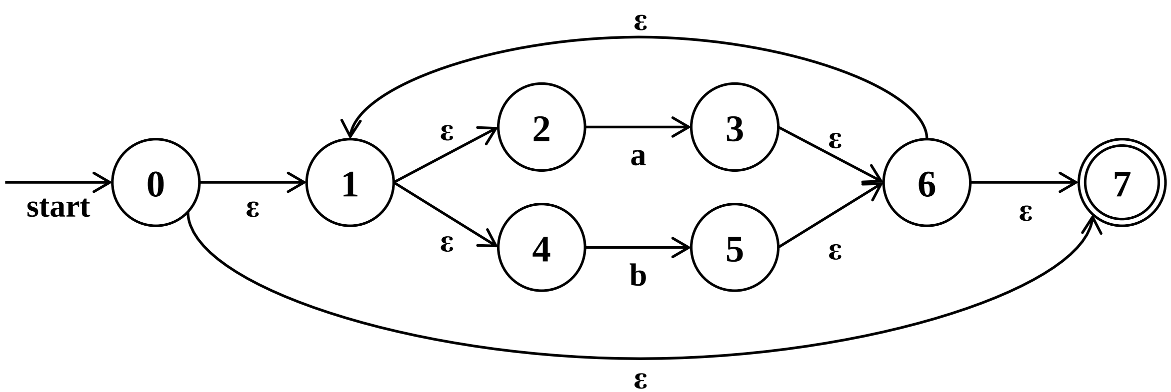
b



a|b, (a|b)



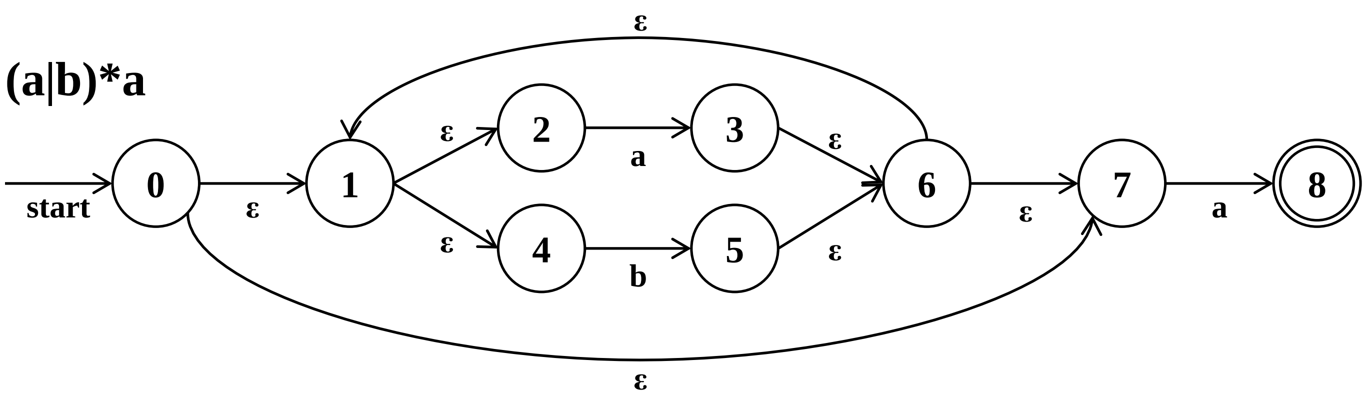
(a|b)*



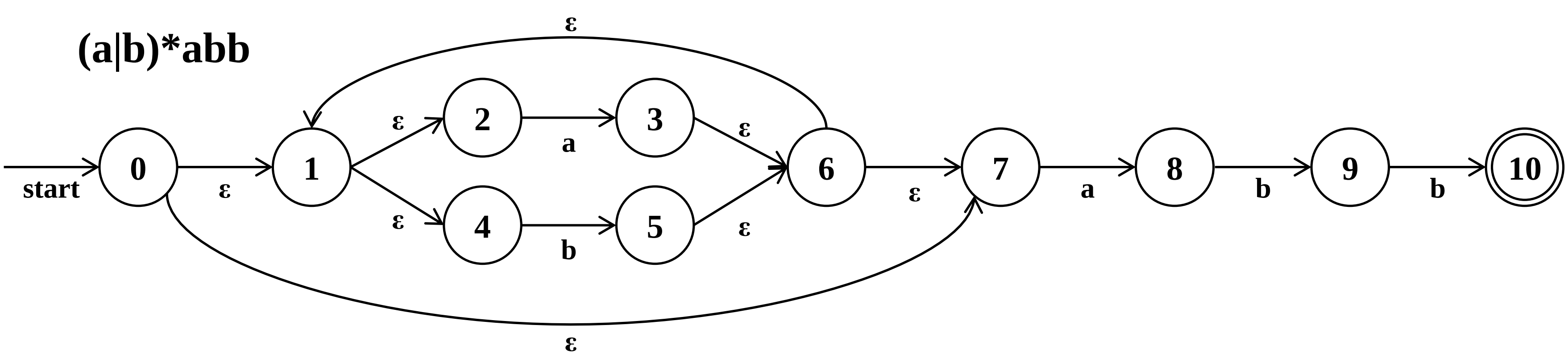
Thompson's Construction Algorithm

- Example: **(a|b)*abb** (cont'd)

(a|b)*a



(a|b)*abb



Thompson’s Construction Algorithm

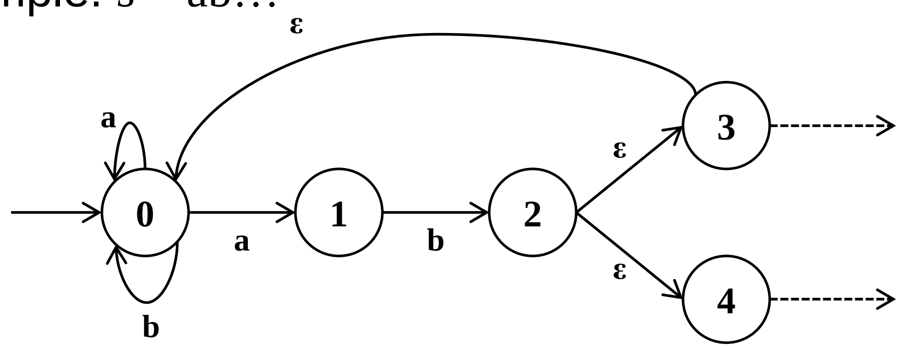
- Properties of NFA constructed by Thompson’s algorithm
  - $N(r)$ has at most twice as many states as there are operators and operands in r
  - $N(r)$ has one start state and one accepting state
  - Each state of $N(r)$ other than the accepting state has either one outgoing transition on a symbol in Σ or up to two outgoing transitions, both on ε.

From NFA to DFA

Conversion of an NFA to a DFA

- Basic idea
  create a state in the DFA that comprises all possible states of the NFA after seeing a certain input string *s*

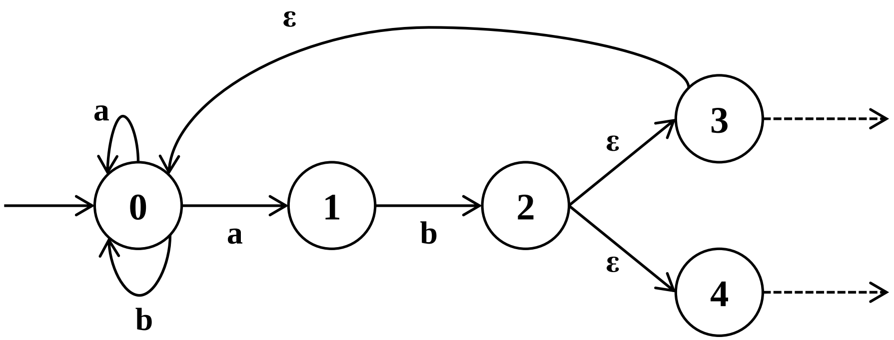
- Example: *s* = ab…



- start: NFA = { 0 } → DFA state 0
- after a: NFA = { 0,1 } → DFA state 01
- after b: NFA = { 2,3,4,0 } → DFA state 0234
- …

Conversion of an NFA to a DFA

- An NFA can be in many different states at any time
  - start: NFA = { 0 }
  - after a: NFA = { 0,1 }
  - after b: NFA = { 2,3,4,0 }
  - …



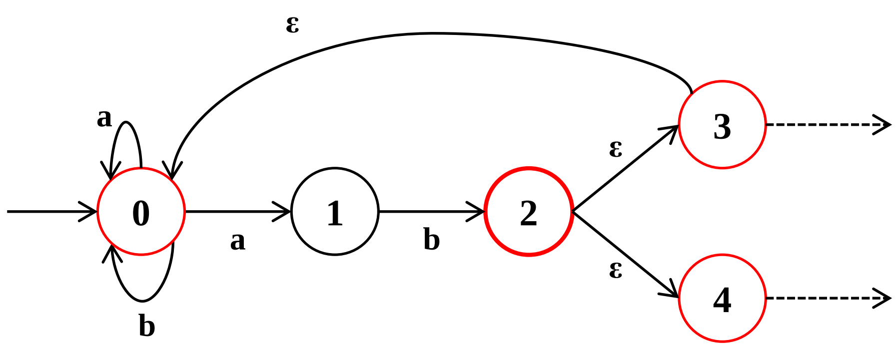
- Maximum number of different state sets
  - NFA with N states
  - subset S
    - |S| ≤ N
    - number of distinct sets: 2^N-1
      - big, but finite → can construct a DFA that simulates the NFA

ε-closure

- ε-closure(s): ε-closure of a state s

  set of NFA states reachable from NFA state s by following only ε-transitions

$s = 2$



- ε-closure(T): ε-closure of a set of states T

  set of NFA states reachable from some NFA state $s \in T$ by following only ε-transitions

CSE 컴퓨터공학부 Department of Computer Science & Engineering

ε-closure

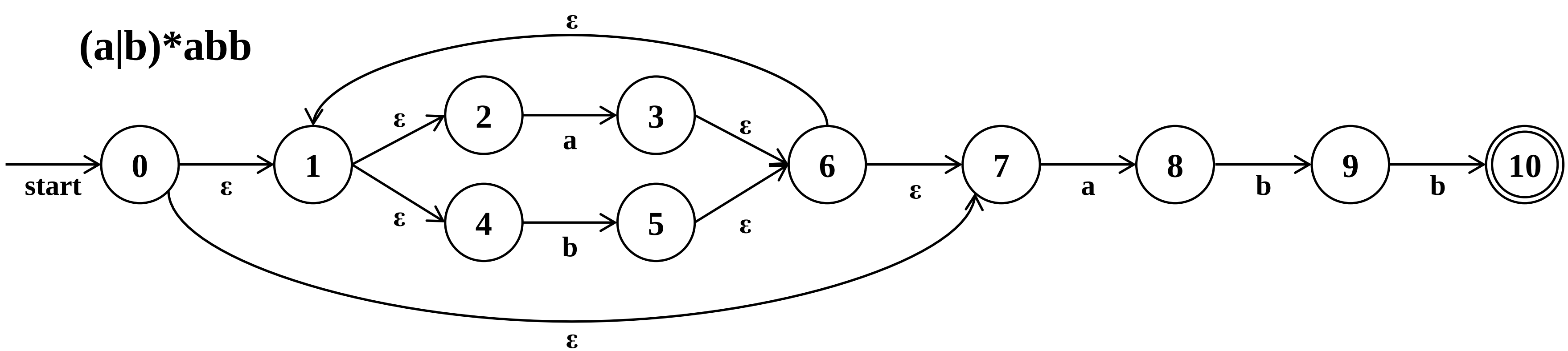
- Computing ε-closure(T):

```
set ε-closure(set T)
{
  stack ← all states in T
  S ← T

  while (!stack.empty()) {
    t = stack.pop()
    for (each state u with an edge t -ε-> u) {
      if (u ∉ S) {
        S ← S ∪ {u}
        stack.push(u)
      }
    }
  }

  return S
}
```

ε-closure

- Example

(a|b)*abb



- ε-closure({0}) = { 0, 1, 2, 4, 7 }
- ε-closure({5}) = { 5, 6, 7, 1, 2, 4 }
- ε-closure({3,9}) = { 1, 2, 3, 4, 6, 7, 9 }

Subset Construction of a DFA from an NFA

- Construct a transition table *Dtran* for DFA D.
  - NFA N
    - states S
    - start state $s_0 \in S$
    - accepting states $F \subseteq S$
    - transition function $f : S \times (\Sigma \cup \varepsilon) \rightarrow S$
  - Operations

Operation	Return value
move(T, a)	set of NFA states to which there is a transition on input symbol a from some state $s \in T$
ε-closure(T)	set of NFA states reachable from some state $s \in T$ on ε-transitions alone

Subset Construction of a DFA from an NFA

- Construct a transition table *Dtran* for DFA *D*.
  - DFA *D*
    - states: *Dstates* = subsets of *S* (except the empty set)
    - start state: ε-closure(s_0)
    - accepting states: all states that include at least one accepting state of *N*
    - transition function: $Dtran : S \times \Sigma \rightarrow S$
  - *Dtran* simulates all possible moves NFA *N* can make on any input string in parallel

$$Dtran(T, a) = \varepsilon\text{-closure}(\text{move}(T, a))$$

Subset Construction of a DFA from an NFA

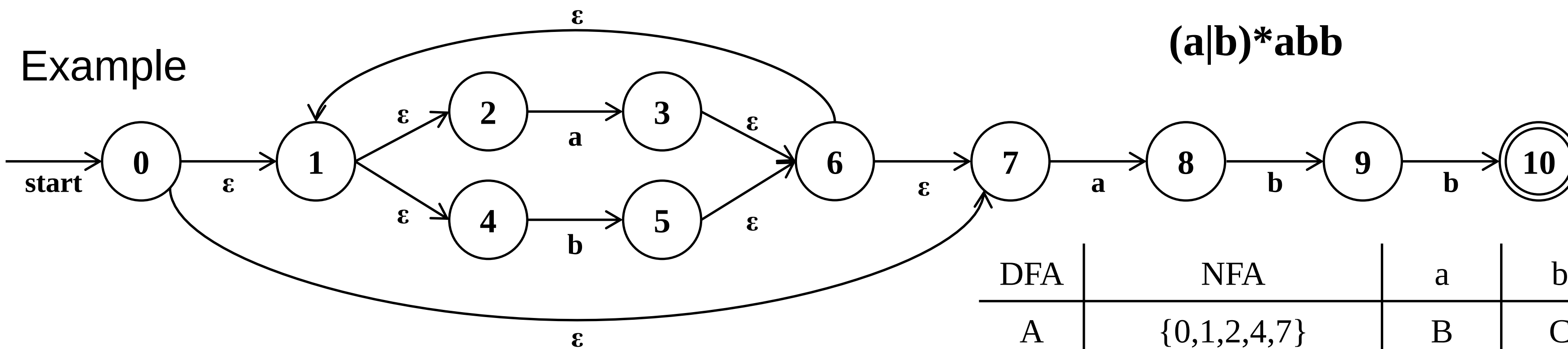
- Subset construction algorithm:

```
Dstates ← {ε-closure(s0)}

while (∃ an unmarked state T in Dstates) {
  mark T;

  for (each input symbol a ∈ Σ) {
    U = ε-closure(move(T, a))
    if (U ∉ Dstates) {
      Dstates ← Dstates ∪ {U}
    }
    Dtrans[T, a] ← U
  }
}
```

Subset Construction of a DFA from an NFA

- Example

(a|b)*abb

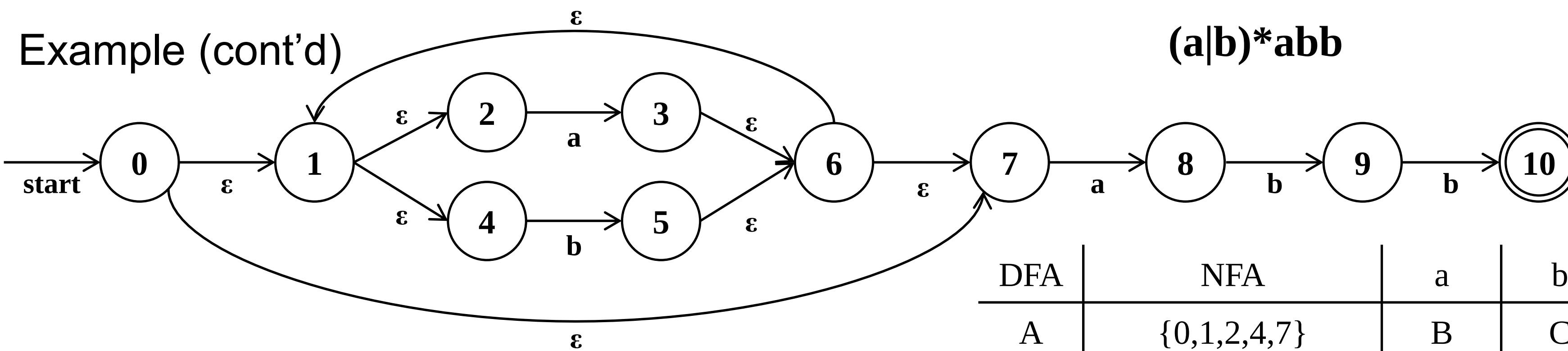


DFA	NFA	a	b
A	{0,1,2,4,7}	B	C
B	{1,2,3,4,6,7,8}	B	D
C	{1,2,4,5,6,7}	B	C

- A = {ε-closure(0)} = {0,1,2,4,7}
- state transitions for A
  - a: ε-closure(move(A,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
  - b: ε-closure(move(A,b)) = ε-closure({5}) = {1,2,4,5,6,7} = C
- state transitions for B
  - a: ε-closure(move(B,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
  - b: ε-closure(move(B,b)) = ε-closure({5,9}) = {1,2,4,5,6,7,9} = D
- state transitions for C
  - a: ε-closure(move(C,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
  - b: ε-closure(move(C,b)) = ε-closure({5}) = {1,2,4,5,6,7} = C

Subset Construction of a DFA from an NFA

- Example (cont'd)



DFA	NFA	a	b
A	{0,1,2,4,7}	B	C
B	{1,2,3,4,6,7,8}	B	D
C	{1,2,4,5,6,7}	B	C
D	{1,2,4,5,6,7,9}	B	E
E	{1,2,4,5,6,7,10}	B	C

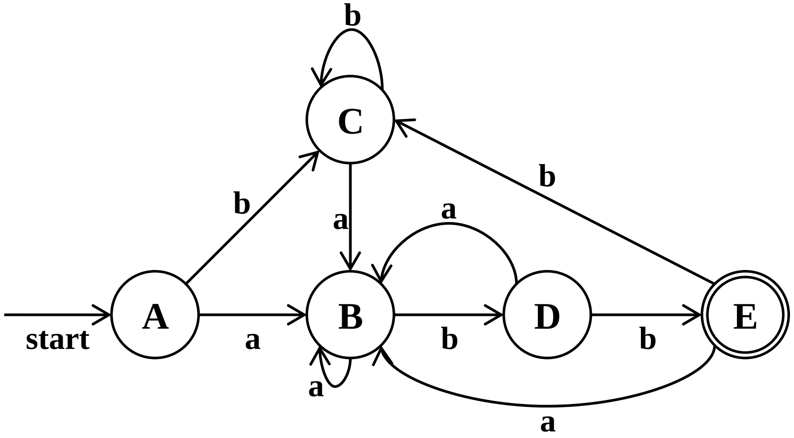
- state transitions for D
  - a: ε-closure(move(D,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
  - b: ε-closure(move(D,b)) = ε-closure({5,10}) = {1,2,4,5,6,7,10} = E
- state transitions for E
  - a: ε-closure(move(E,a)) = ε-closure({3,8}) = {1,2,3,4,6,7,8} = B
  - b: ε-closure(move(E,b)) = ε-closure({5}) = {1,2,4,5,6,7} = C

Subset Construction of a DFA from an NFA

- Example (cont'd)
  - Dstates = { A, B, C, D, E }
  - Dtrans

DFA	NFA	a	b
A	{0,1,2,4,7}	B	C
B	{1,2,3,4,6,7,8}	B	D
C	{1,2,4,5,6,7}	B	C
D	{1,2,4,5,6,7,9}	B	E
E	{1,2,4,5,6,7,**10**}	B	C

DFA for **(a|b)*abb**

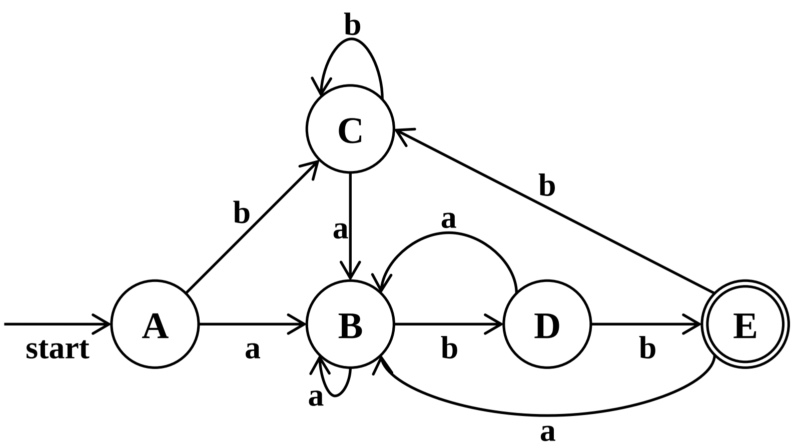


Minimizing the Number of States of a DFA

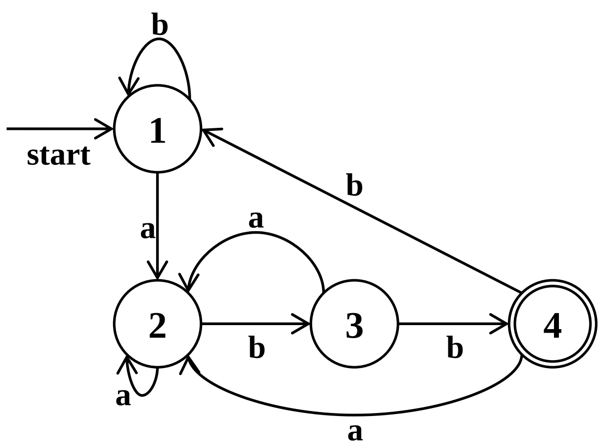
Minimizing the Number of States of a DFA

- The subset construction algorithm may lead to DFAs with superfluous states

DFA for **(a|b)*abb**
(subset construction)

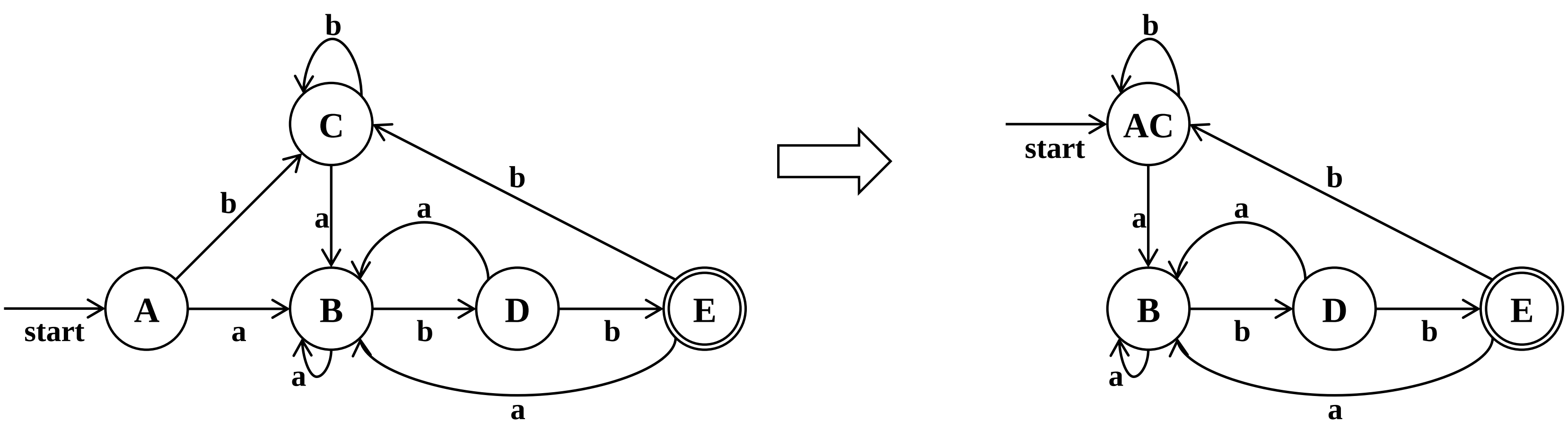


minimum state DFA for **(a|b)*abb**



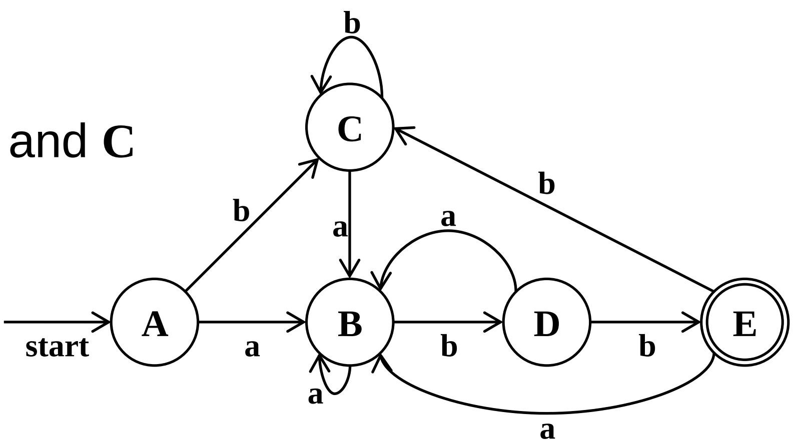
Minimum State DFA

- For any regular language, there is a unique minimum state DFA
- Construction of the minimum state DFA is possible from *any* DFA for the same language
- Idea: merge functionally equivalent states



Distinguishable States

- Distinguishable and undistinguishable states
  - string x distinguishes state s from state t if only one of the states reached from s and t by following the path with label x is an accepting state.
  - examples:
    - string **ba** does not distinguish states **A** and **C**
    - string **bb** distinguishes states **B** and **C**
    - the empty string ε distinguishes accepting from non-accepting states



  - state s is distinguishable from state t if there is some string that distinguishes the two states.

State-Minimization Algorithm

- State-minimization algorithm
  - partition the states of a DFA into groups of undistinguishable states
  - merge each group into a single state
  - these states make up the minimum-state DFA

  - input: original DFA D
  - output: minimum-state DFG D'
  - operation
    - initial partition: { accepting states of D }, { non-accepting states of D }
    - step: if an input a distinguishes a group in the current partition, split the group into smaller subgroups until for no group and no input symbol the group is split further

State-Minimization Algorithm

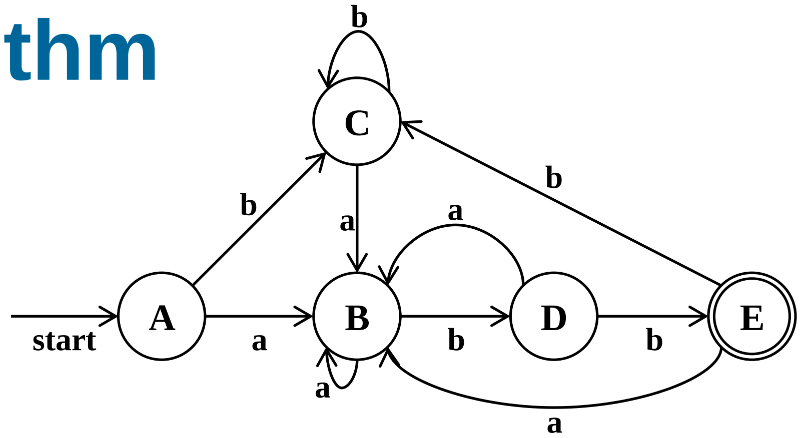
- Partitioning

```
Π ← { S-F, F }

do {
  Π_new ← Π
  for (each group G of Π) {
    partition G into subgroups such that two states s and t
        are in the same subgroup iff ∀a∈Σ, the transitions
        for states s and t on a go to states in the same
        group in Π
    replace G in Π_new by the set of all subgroups
  }
} while (Π_new ≠ Π)
```

State-Minimization Algorithm

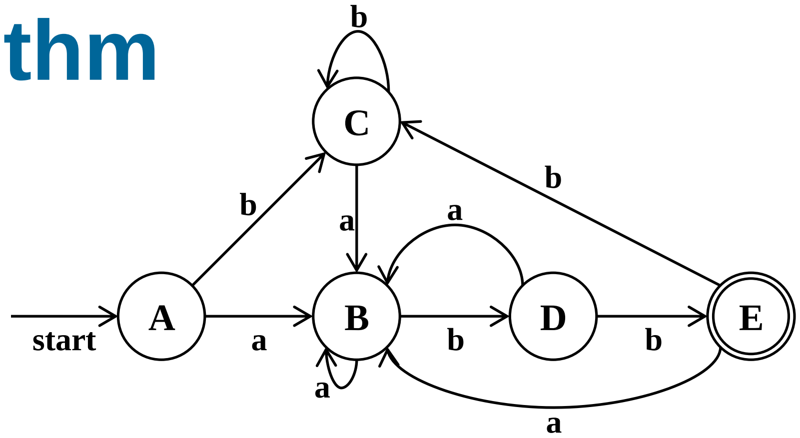
- Generating states
  - for each group in Π choose one state as the representative for that group. Each representative forms a state in the minimum-state DFA D'
  - start state of D'
    representative of the group containing the start state of D
  - accepting states of D'
    representative of group(s) containing an accepting state of D
  - transitions of D'
    let s be the representative of some group G in Π. In D, state s has a transition on input a to state t; t is represented by r in D'. Then there is a transition from s to r on a in D'.

State-Minimization Algorithm



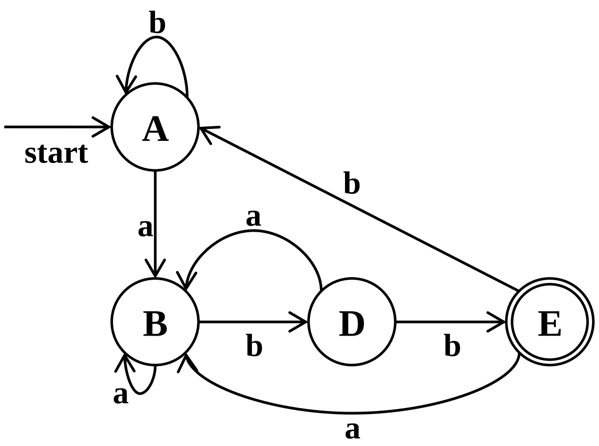
- Example
  - initial partition
    $\Pi \leftarrow \{ \{A,B,C,D\}, \{E\} \}$
  - ignore {E} (|{E}| = 1 and can thus not be split further)
  - for {A,B,C,D}
    - input a: all states go to B
      → a does not distinguish A,B,C,D on a
    - input b: {A,C} go to C, B goes to D (both in {A,B,C,D}); D goes to E
      → split into {A,B,C} and {D}
  - ignore {D}, for {A,B,C}
    - input a: all states go to B
    - input b: {A, C} go to C; B goes to D
      → split into {A,C},{B}
  - ignore {B}, for {A,C}
    - input a: both states go to B
    - input b: both states go to C

State-Minimization Algorithm



- Example
  - final partition
    $\Pi \leftarrow \{ \{A,C\}, \{B\}, \{D\}, \{E\} \}$
  - pick representatives: A for {A,C}; B, D, E for themselves
  - start state: A in D → A in D'
  - accepting states: E in D → E in D'
  - transitions:

minimum-state DFA	**a**	**b**
A	B	A
B	B	D
D	B	E
E	B	A



Simulating an NFA

Simulating an NFA

- Instead of first constructing a DFA, we can directly simulate an NFA

```
set of int NFA(char *input, int s0, int (*move)(set of int, char))
{
  set of int S = ε-closure(s0);
  char c = *input++;
  while (c != '\0') {
    S = ε-closure(move(S, c));
    c = *input++;
  }
  return S
}
```

```
set of int mtab[S][Σ];

set of int move(set of int s,
                char c)
{
  …
}
```

usage:

```
set of int f = NFA("aababb", 0, m);
if (f ∩ F) printf("accept");
else printf("nope");
```

Comparison with DFA Simulation

- DFA simulation vs NFA simulation

```
int DFA(char *input, int s0,
      int (*move)(int, char))
{
  int s = s0;
  char c = *input++;
  while (c != '\0') {
    s = move(s, c);
    c = *input++;
  }
  return s
}

int move(int s, char c);
```

```
set of int NFA(char *input, int s0,
      int (*move)(set of int, char))
{
  set of int S = ε-closure(s0);
  char c = *input++;
  while (c != '\0') {
    S = ε-closure(move(S, c));
    c = *input++;
  }
  return S
}

set of int move(set of int s,
              char c);
```

Comparison with DFA Simulation

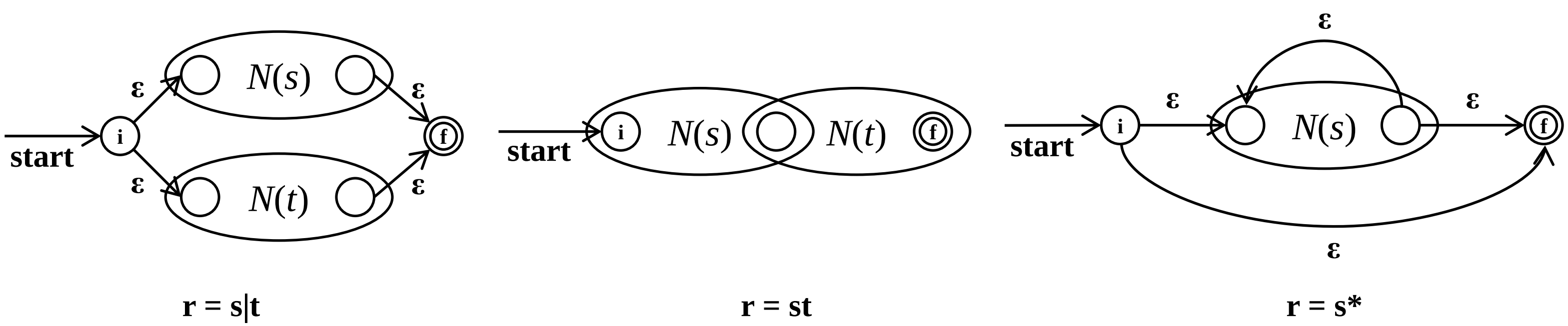
- Cost analysis
  - NFA
    - conversion of a regular expression r to an NFA: $O(|r|)$
    - simulation: $O((|n|+|m|) \times |x|) = O(|r| \times |x|)$
      (n states, m transitions; observe that $n \leq |r|$ and $m \leq 2|r|$)
  - DFA
    - subset construction: $O(|r|^3)$ in the typical, $O(|r|^2 2^{|r|})$ in the worst case
    - simulation: $O(|x|)$

Automaton	Initial	Per String
NFA	$O(\|r\|)$	$O(\|r\| \times \|x\|)$
DFA typical case	$O(\|r\|^3)$	$O(\|x\|)$
DFA worst case	$O(\|r\|^2 2^{\|r\|})$	$O(\|x\|)$

Recap: RE → NFA → DFA → minimal DFA

- RE → NFA: Thompsons Construction Algorithm
  - base rules:

  - composition rules:



Recap: RE → NFA → DFA → minimal DFA

- NFA → DFA: Subset Construction Algorithm
  - idea: states of DFA = state set of NFA

```
Dstates ← {ε-closure(s0)}

while (∃ an unmarked state T in Dstates) {
  mark T;

  for (each input symbol a ∈ Σ) {
    U = ε-closure(move(T, a))
    if (U ∉ Dstates) {
      Dstates ← Dstates ∪ {U}
    }
    Dtrans[T, a] ← U
  }
}
```

```
set ε-closure(set T)
{
  stack ← all states in T
  S ← T

  while (!stack.empty()) {
    t = stack.pop()
    for (each state u with an edge t → u){
      if (u ∉ S) {
        S ← S ∪ {u}
        stack.push(u)
      }
    }
  }

  return S
}
```

Recap: RE → NFA → DFA → minimal DFA

- DFA → minimal DFA: State Minimization Algorithm
  - idea: merge states that behave identical for all possible input symbols

```
Π ← { S-F, F }

do {
  Π_new ← Π
  for (each group G of Π) {
    partition G into subgroups such that two states s and t
        are in the same subgroup iff ∀a∈Σ, the transitions
        for states s and t on a go to states in the same
        group in Π
    replace G in Π_new by the set of all subgroups
  }
} while (Π_new ≠ Π)
```

+ some renaming & fixups after partitioning

That's all very nice, but how do I implement it?

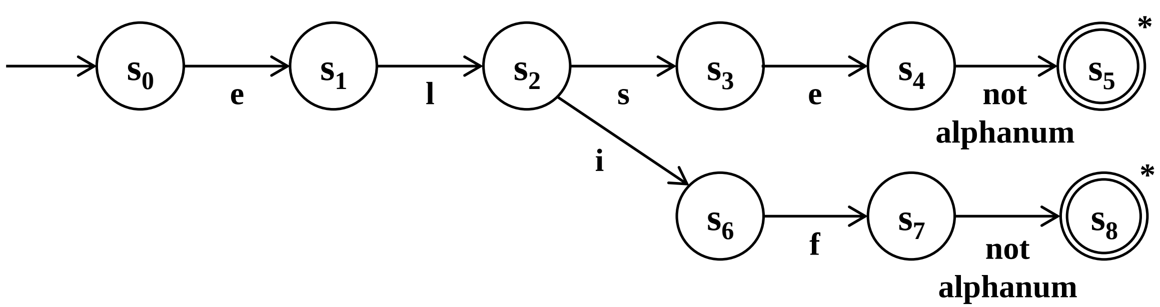
CSE 컴퓨터공학부
Department of Computer Science & Engineering

DFA & Lookahead

- Example:
  - language syntax:

    `keywords → else | elif`

  - transition diagram



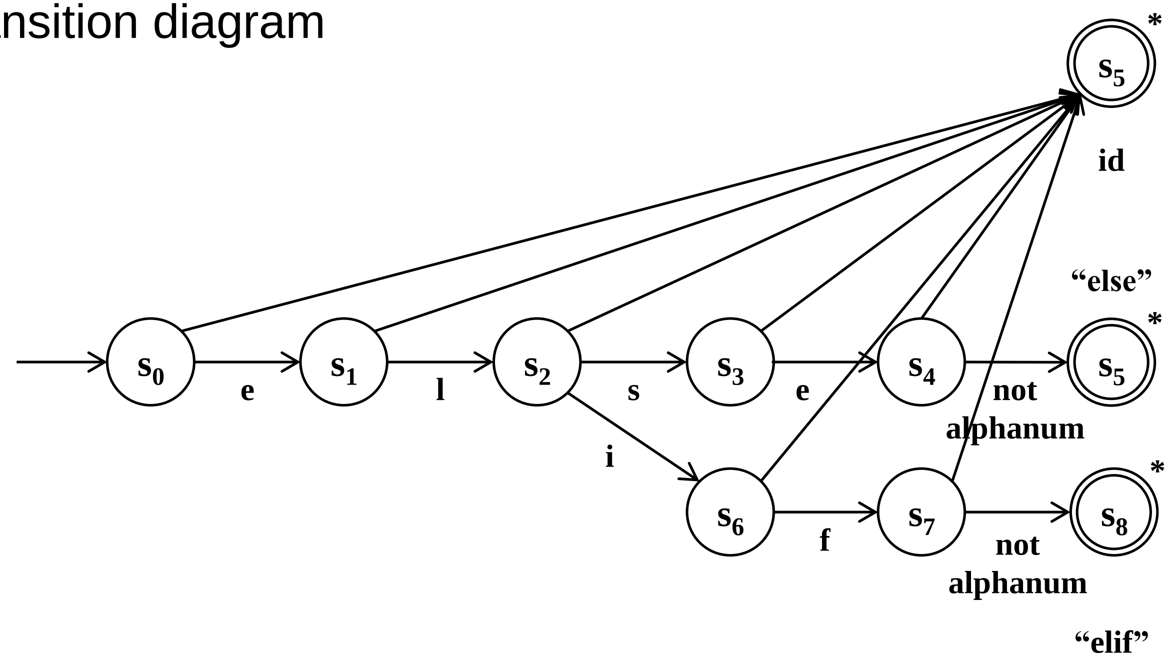
* = retract input by one character

DFA & Lookahead

- Example:
  - language syntax:

```
keywords  → else | elif
id        → alph (alphanum)*
```

  - transition diagram

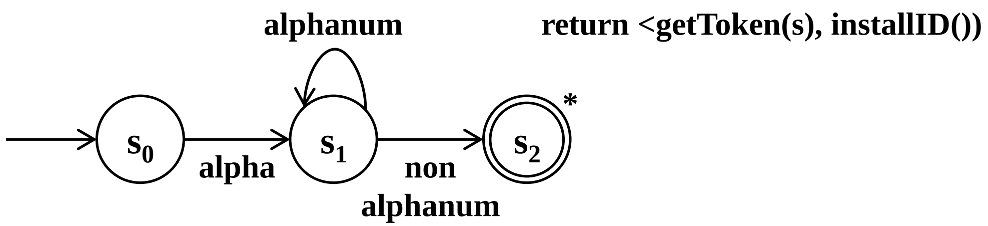


DFA & Lookahead

- Example:
  - language syntax:

```
keywords   → else | elif
id         → alph (alphanum)*
```

  - transition diagram



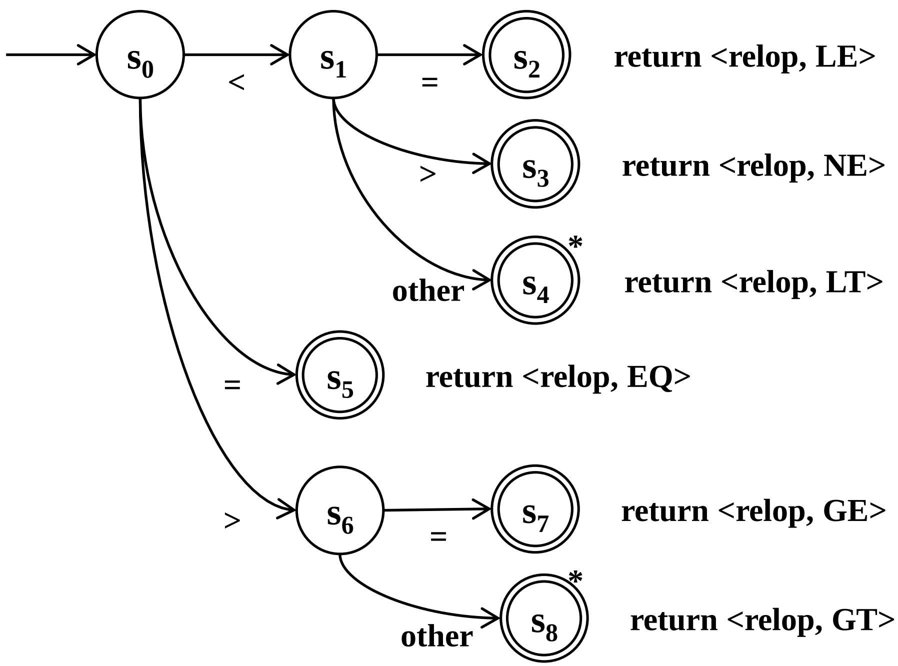
plus install keywords in symbol table

DFA & Lookahead

- Example:
  - language syntax:

    `relop → < | <= | > | >= | = | <>`

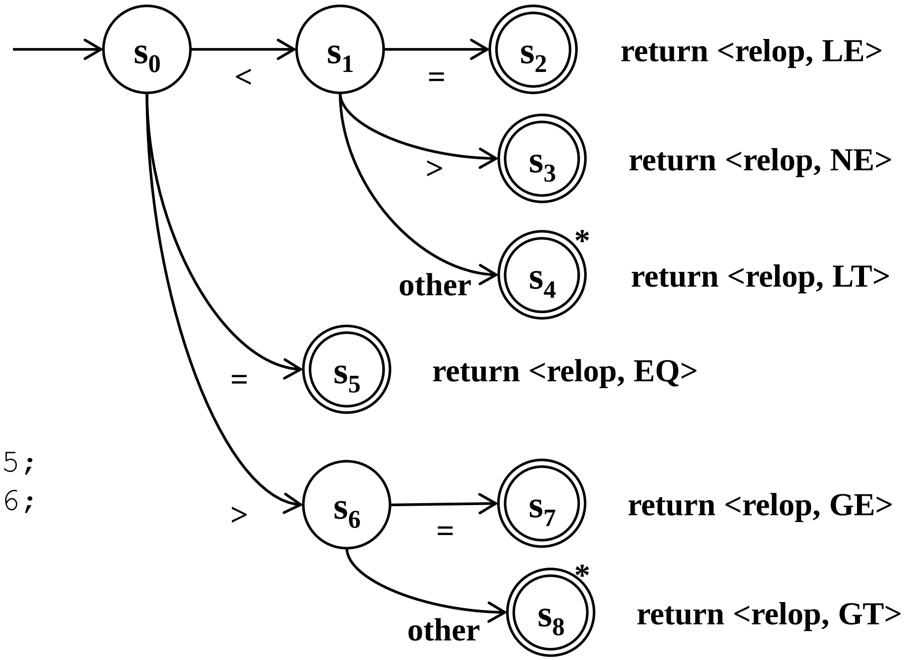
  - transition diagram



DFA & Lookahead

- Direct Implementation

```
TOKEN getRelop()
{
  TOKEN retToken = new(RELOP);

  while (1) {
    switch (state) {
      case 0: c = nextChar();
              if (c == '<') state = 1;
              else if (c == '=') state = 5;
              else if (c == '>') state = 6;
              else fail();
              break;

      case 5: retToken.attribute = EQ;
              return retToken;

      case 6: c = nextChar();
              if (c == '=') state = 7;
              else state = 8;
              break;

      case 8: retract();
              retToken.attribute = GT;
              return retToken;
...
```



Lexical Analysis with Flex

Lexical Analysis with Flex

- see “Lexical Analysis with Flex”, Vern Paxson et al (on eTL)

- Example

```
%option noyywrap

%%

"<"               { printf("tRelOp (<)\n"); }
"<="              { printf("tRelOp (<=)\n"); }
">"               { printf("tRelOp (>)\n"); }
">="              { printf("tRelOp (%s)\n", yytext); }
"="               { printf("tRelOp (%s)\n", yytext); }
"<>"              { printf("tRelOp (%s)\n", yytext); }

%%

int main(void)
{
  yylex();
  return 0;
}
```

Lexical Analysis with Flex

- Example

```
$ flex RelOp.l
$ gcc -o relop lex.yy.c
$ echo "<<<==>=><>" | ./relop
tRelOp (<)
tRelOp (<)
tRelOp (<=)
tRelOp (=)
tRelOp (>=)
tRelOp (>)
tRelOp (<>)

$
```

Lexical Analysis with Flex

- lex.yy.c

```
…
static yyconst flex_int16_t yy_accept[12] =
    {   0,
        0,    0,    8,    7,    1,    5,    3,    2,    6,    4,
        0
    } ;

static yyconst flex_int32_t yy_ec[256] =
    {   0,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
        …
    } ;

static yyconst flex_int32_t yy_meta[5] =
    {   0,
        1,    1,    2,    1
    } ;

static yyconst flex_int16_t yy_base[13] =
    {   0,
        0,    0,    8,    9,    2,    9,    0,    9,    9,    9,
        9,    5
    } ;
```

Lexical Analysis with Flex

- lex.yy.c (cont'd)

```
…
static yyconst flex_int16_t yy_def[13] =
    {   0,
       11,    1,   11,   11,   11,   11,   12,   11,   11,   11,
        0,   11
    } ;

static yyconst flex_int16_t yy_nxt[14] =
    {   0,
        4,    5,    6,    7,    8,    9,   10,   11,    3,   11,
       11,   11,   11
    } ;

static yyconst flex_int16_t yy_chk[14] =
    {   0,
        1,    1,    1,    1,    5,    5,   12,    3,   11,   11,
       11,   11,   11
    } ;
…
```

Implementation from Scratch

Implementation from Scratch

- Often, the lexical structure is easy enough to be implemented without an explicit DFA

  - regular expressions
    - ident: **ident = letter (letter | digit)***
    - number: **number = digit (digit)***
    - operators: **operator = '+', '-'**
    - rel. operators: **relop = '<', '<='**

  - assume C++ input streams

```
std::istream in
```

    - `get()`: return next character in input stream
    - `peek()`: look at next character in input stream (without removing it)

Implementation from Scratch

- Skeleton

```
Token* Scanner::Get()
{
  EToken token = tUndefined;
  string lexeme = "";
  char c;

  while (IsWhite(in.peek()) in.get();

  c = in.get();
  lexeme = c;

  switch (c) {
    ...
  }

  return new Token(token, lexeme);
}
```

initialize token and lexeme

skip over white space

consume next character

state machine (next slide)

return token

Implementation from Scratch

- Scanning the next token
  - group lexemes that may start with the same letter
  - consume characters as long as the proper prefix is identical
  - use peek() to distinguish the end of a token vs. longer tokens (e.g., "<" vs "<=")

```
switch (c) {
  case '+':
  case '-':
    token = tOperator;
    break;

  case '<':
    token = tRelOp;
    if (in.peek() == '=') lexeme += in.get();
    break;

  ...
```

Implementation from Scratch

- Scanning the next token (cont'd)

```
switch (c) {
  …
  default:
    if (IsLetter(c)) {
      token = tIdent;
      while ((IsLetter(in.peek()) || IsDigit(in.peek()))
        lexeme += in.get();
    } else if (IsDigit(c)) {
      token = tNumber;
      while (IsDigit(in.peek()) lexeme += in.get();
    } else {
      token = tError;
    }
}
```

Implementation from Scratch

- Dealing with keywords
  - regular expressions
    - ident: **ident = letter (letter | digit)***
    - number: **number = digit (digit)***
    - operators: **operator = '+', '-'**
    - rel. operators: **relop = '<', '<='**
    - keywords: **keyword = 'begin' | 'end' | 'while' | …**
  - store keywords in a map containing the keyword and the token type

```
map<string, EToken> keyword;
```

and initialize the table with the keywords and their token type

```
keyword['begin'] = tBegin;
keyword['end'] = tEnd;
…
```

Implementation from Scratch

- Dealing with keywords
  - keywords conform to identifiers, so all we need to do is check every time we have seen an identifier:

```
switch (c) {
  …
  default:
    if (IsLetter(c)) {
      while ((IsLetter(in.peek()) || IsDigit(in.peek()))
        lexeme += in.get();

      map<string, EToken>::iterator it = keyword.find(lexeme);
      if (it != keyword.end()) token = (*it).second;
      else token = tIdent;
    } …
}
```