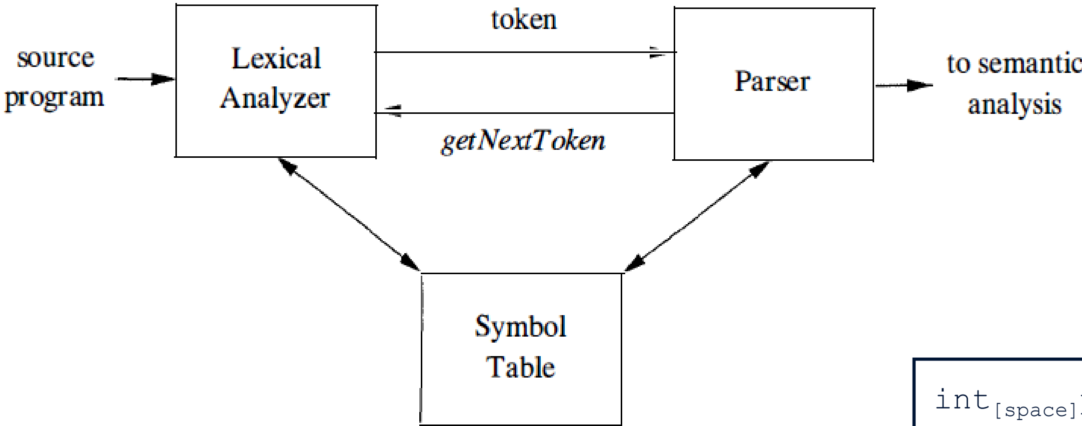
CS325 – Compiler Design

Lexing (Scanning)

Dr. Gihan R. Mudalige
g.mudalige@warwick.ac.uk

WARWICK
THE UNIVERSITY OF WARWICK

- Recall from last lecture that Lexing (also known as scanning) transform a stream of characters into a stream of words (also known as tokens) in some language.




```
int pos;
int init;
int rate;
main() {
  pos = init + rate * 60;
}
```

- The Lexer sees a sequence of characters as input

```
int[space]pos;[newline]int[space]init;[newline]int[space]rate;[newline]main()[newline]{[newline]pos[space]=[space]init[space]+[space]rate[space]*[space]60;[newline]}[newline][eof]
```

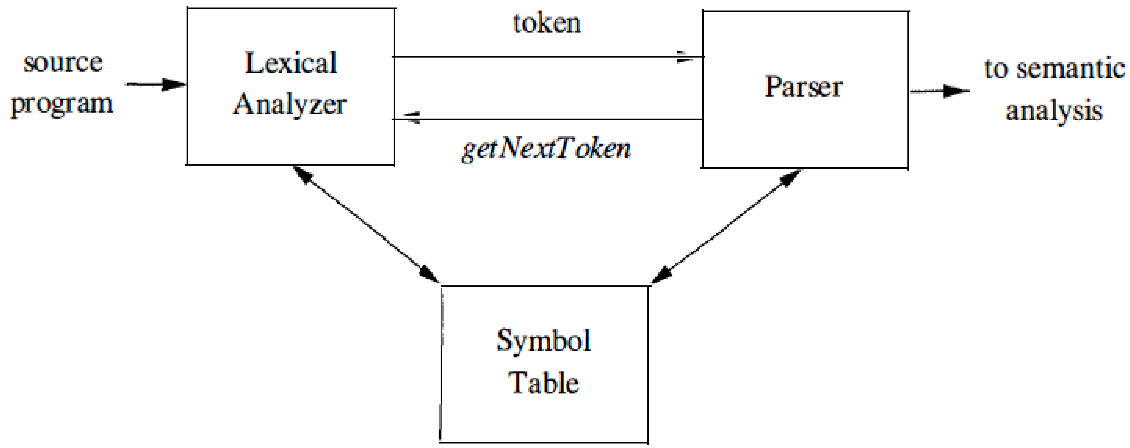
- And outputs a sequence of tokens
- and a symbol table

```
<keyword,int>, <id,1>, <";">,
<keyword,int>, <id,2>, <";">,
<keyword,int>, <id,3>, <";">,
<id,main>, < "(" >, < ")" >,
<"{">, <id,1>, <op,"=">, <id,2>,
<op,"+">, <id,3>, <op,"*">,
<num, 4>, <";">, <"}">, <eof>
```

1	pos	...
2	init	...
3	rate	...
4	num	60

Symbol table

LEXING

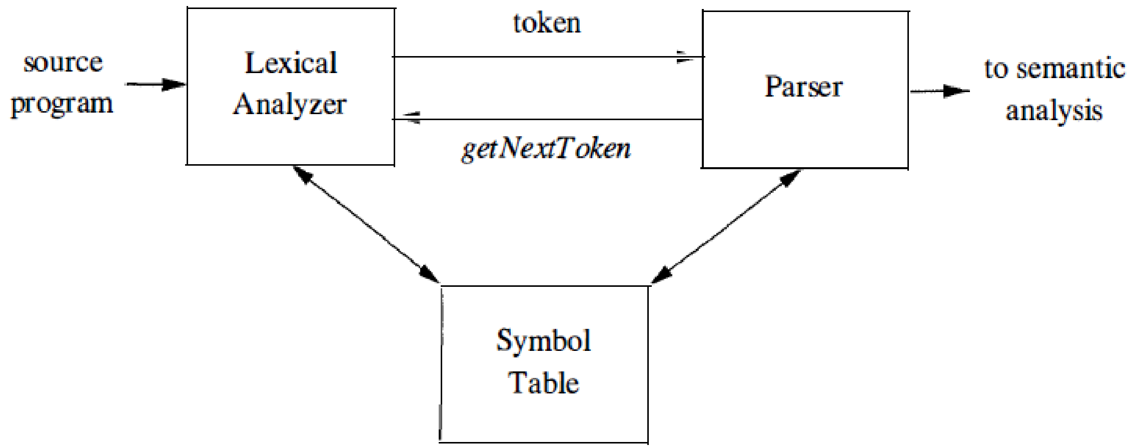


The Lexical analyser will :

- ❑ Scan the input one character at a time
- ❑ Remove white space (blank, newline, tab, etc.)
- ❑ Groups the characters into meaningful sequences called lexemes
- ❑ For each lexeme, produces as output a token of the form: *<token-name, optional attribute-value>*
- ❑ Do error report/recovery

- ❑ The stream of tokens is sent to the parser for syntax analysis
- ❑ *getNextToken* – get the lexer to read characters from its input until it can identify the next lexeme and produce for it the next token, which is returned to the parser

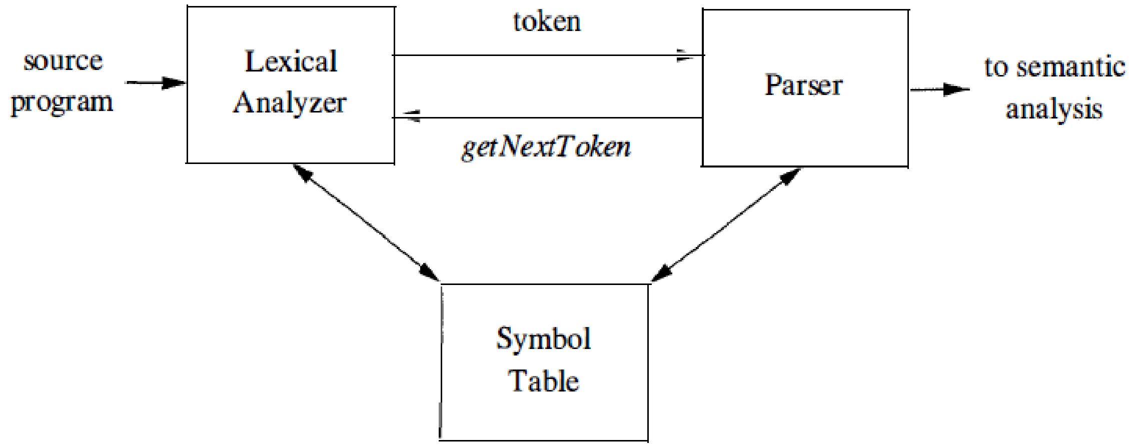
LEXING



- Additionally the lexical analyzer interacts with the symbol table
- When a lexeme constituting an identifier is detected, that lexeme is entered into the symbol table
- Sometimes the symbol assists in determining the correct token to be passed to the parser

- Lexical errors – misspellings of identifiers, keywords, or operators and missing quotes around text intended as a string

LEXING – SOME TERMINOLOGY



- **Token** - a pair consisting of a token name and an optional attribute value, *<name, opt-attrib-val>*
  - Token name is an abstract symbol representing a kind of lexical unit (e.g. keyword, identifier)
  - Attribute value - e.g. token **number** matches both 0 and 1, then the attributes of the token are 0 and 1

- **Pattern** - description of the form that the lexemes of a token may take
  - For a keyword (e.g. `if`, `else`, `while`) as a token, the pattern is just the sequence of characters that form the keyword
  - For identifiers and some other tokens, the pattern is a more complex structure that is matched by many strings

- **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.

EXAMPLES OF TOKENS

TOKEN	INFORMAL DESCRIPTION	SAMPLE LEXEMES
if	characters `i`, `f`	`if`
else	characters `e`, `l`, `s`, `e`	`else`
comparison	`<` or `>` or `<=` or `>=` or `==` or `!=`	`<=`, `!=`
id	letter followed by letters and digits	`pi`, `score`, `D2`
number	any numeric constant	`3.14159`, `0`, `6.02e23`
literal	anything but `"`, surrounded by `"`'s	`"core dumped"`

```
printf("Total = %d\n", score);
```

- **`printf`** and **`score`** are lexemes matching the pattern for token **id** (identifier)
- **`"Total = %d\n"`** is a lexeme matching **literal**

- The pattern for token **number** matches both `0` and `1`, in this case the lexer returns the token together with an attribute that describes the lexeme that was matched

< **number**, 0> and < **number**, 1>

- The token name influences parsing decisions (in syntax analysis),
- The attribute value influences translation of tokens after the parse (in semantic analysis)

Token attributes and Symbol table entries

- Operators, punctuation, and keywords usually do not need an attribute value
- Matching identifiers, **id**s, usually get an entry into the symbol table and a pointer to that entry as the attribute of the token

```
E = M * C ** 2
```

- For the Fortran expression above, we get the following tokens:

<**id**, pointer to symbol-table entry for `E`>
<**assign_op**>
<**id**, pointer to symbol-table entry for `M`>
<**mult_op**>
<**id**, pointer to symbol-table entry for `C`>
<**exp_op**>
<**number**, integer value 2>

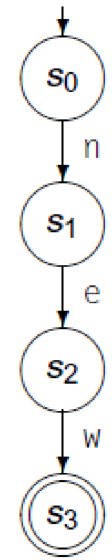
Lexing - Overview

- Recognizers - program that identifies words in a stream of characters
- Regular expressions - a formal notation for specifying syntax
- Stepwise approach to converting regular expressions into a recognizer

RECOGNIZER – VERY (VERY) SIMPLE EXAMPLE

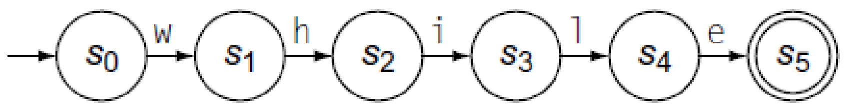
- Recognizer for identifying the key word “new”

```
c ← NextChar();
if (c = 'n')
   then begin;
      c ← NextChar();
      if (c = 'e')
          then begin;
            c ← NextChar();
            if (c = 'w')
                then report success;
                else try something else;
         end;
         else try something else;
   end;
   else try something else;
```

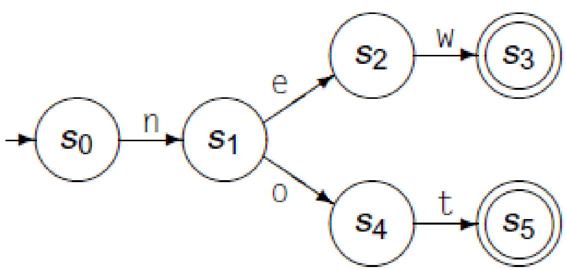


- Assume that *NextChar( )* returns the next character
- The code simply tests for ‘n’ followed by ‘e’ followed by ‘w’
- The transition diagram to the left diagrammatically shows this recognizer
- s_0 - Start state and s_3 - accepting state

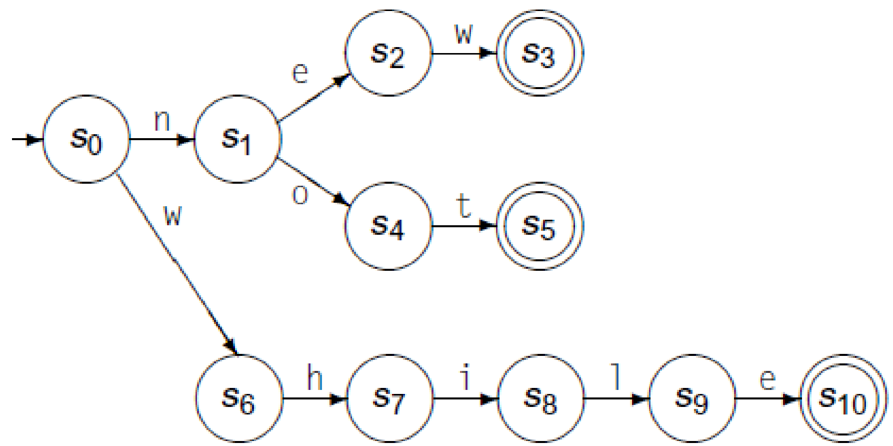
- Recognizer for identifying the key word "while"



- We can combine states to recognize multiple words



Recognizer for "new" and "not"



Recognizer for "new", "not" and "while"

Finite automatons

- The transition diagrams serve as abstractions for the recognizers
- They can also be viewed as formal mathematical objects, called finite automata, that specify recognizers

A formalism for recognizers that has a finite set of states, an alphabet, a transition function, a start state, and one or more accepting states

- Formally a Finite automata (FA) is given by a five-tuple (S, Σ, δ, s_0, S_A) where

S – Finite set of states in the recognizer along with an error state s_e

Σ – Finite alphabet used by the recognizer. (Typically the union of edge labels in the transition diagram)

$\delta(s,c)$ – Recognizer's transition function. Maps each state $s \in S$ and each character $c \in \Sigma$ into some next state. In state s_i with input character c, the FA takes the transition

$$s_i \xrightarrow{c} \delta(s_i, c)$$

s_0 – Start state

S_A – The set of accepting states, $S_A \subseteq S$. Each state in S_A appears as a double circle in the transition diagram.

The FA for new or not or while

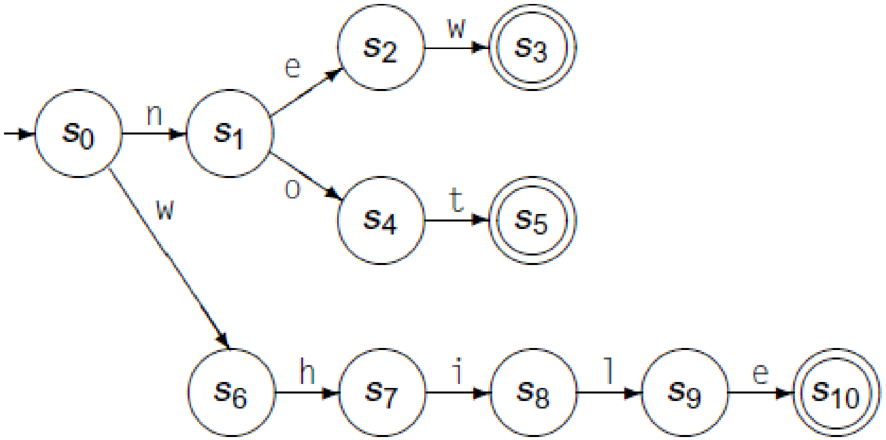
Set of states $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$

Alphabet $\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{l}, \mathtt{n}, \mathtt{o}, \mathtt{t}, \mathtt{w}\}$

Transition function
$$\delta = \left\{ \begin{array}{lllll} s_0 \xrightarrow{\mathtt{n}} s_1, & s_0 \xrightarrow{\mathtt{w}} s_6, & s_1 \xrightarrow{\mathtt{e}} s_2, & s_1 \xrightarrow{\mathtt{o}} s_4, & s_2 \xrightarrow{\mathtt{w}} s_3, \\ s_4 \xrightarrow{\mathtt{t}} s_5, & s_6 \xrightarrow{\mathtt{h}} s_7, & s_7 \xrightarrow{\mathtt{i}} s_8, & s_8 \xrightarrow{\mathtt{l}} s_9, & s_9 \xrightarrow{\mathtt{e}} s_{10} \end{array} \right\}$$

Start state $s_0 = s_0$

Accepting States $S_A = \{s_3, s_5, s_{10}\}$



Alphabet, Word and Language

- An Alphabet is any finite set of symbols.

  e.g. The set *{0, 1}* is the binary alphabet, ASCII symbols (128 symbols), Unicode (~ 100k symbols)

- A String over and alphabet is a finite sequence of symbols drawn from that alphabet.

The terms Sentence and Word are often used as synonyms for String

The empty string, denoted by ε, is the string of length zero.

- A language is any countable set of strings over some fixed alphabet. The definition of language does not require that any meaning be ascribed to the strings in the language.

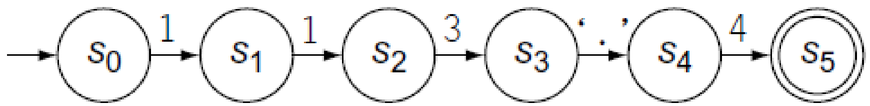
- A Language is defined using grammars – this is checked during parsing (i.e. syntax analysis)
- But, identifying the words that belong to that language is done by a recognizer (i.e. lexing)

- We need to be able to check if any string on the alphabet is a member of the language.

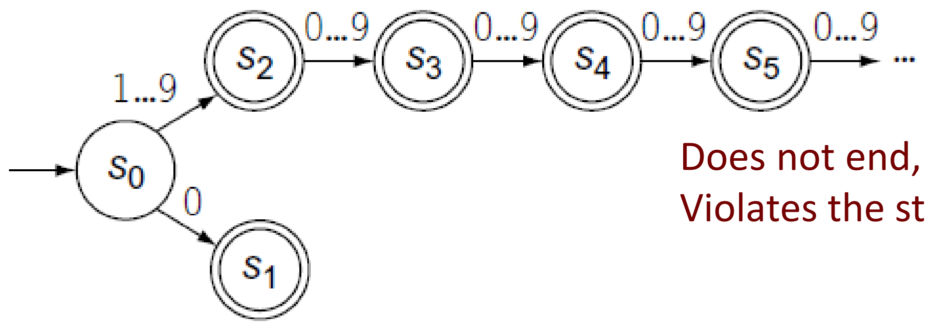
  The way this is done is by describing a recognizing automaton

Recognizer for more complex words

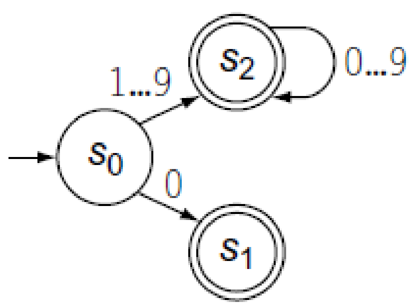
- But could we recognize a number with such a recognizer ?
- A specific number such as 113.4 is easy



- But to be useful, we need a transition diagram that can recognize any number
- Transition diagram for unsigned integers

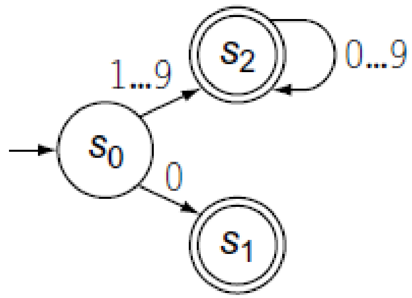


Does not end,
Violates the stipulation that *S* is finite



Simplify using cycles

A RECOGNIZER FOR UNSIGNED INTEGERS



$S = \{s_0, s_1, s_2, s_e\}$

$\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$$\delta = \left\{ \begin{array}{ll} s_0 \xrightarrow{0} s_1, & s_0 \xrightarrow{1-9} s_2 \\ s_2 \xrightarrow{0-9} s_2, & s_1 \xrightarrow{0-9} s_e \end{array} \right\}$$

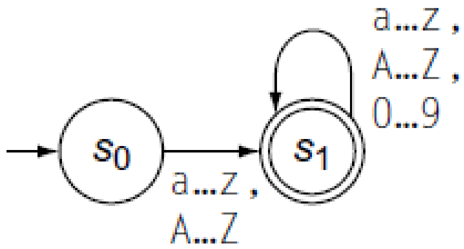
$S_A = \{s_1, s_2\}$

```
char ← NextChar( );
state ← s0 ;

while (char ≠ eof and state ≠ se) do
   state ← δ(state,char);
   char ← NextChar( );
end;

if (state ∈ SA )
    then report acceptance;
    else report failure;
```

δ	0	1	2	3	4	5	6	7	8	9	**Other**
s_0	s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_1	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_e
s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e	s_e

A RECOGNIZER FOR IDENTIFIER NAMES

- A simplified version of the rule that governs identifier names in Algol-like languages (C, C++, Java)
- An identifier consists of an alphabetic character followed by zero or more alphanumeric characters.



- Many programming languages extend the notion of alphabetic character to include designated special characters, such as the underscore.

- As you can see we can represent character-by-character scanners with a transition diagram
- That diagram, in turn, corresponds to a finite automaton.
- Small sets of words are easily encoded in acyclic transition diagrams.
- Infinite sets require cyclic transition diagrams.

Regular Expressions

- FAs (and the transition diagrams they represent) are not particularly concise specifications.
- For an efficient scanner implementation, a concise notation is required and a way of turning those specifications into an FA and into code that implements the FA.
- This *notation* is provided by Regular Expressions (REs)

You have all seen REs before …. but lets start with some very simple examples:

- RE for language with single word *new* will be new (i.e. the same spelling)
- RE for language with only two words *new* and *not* can be n(ew|ot)

- The RE for unsigned integers :

0|(1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)*

An unsigned integer is either a zero, or a digit that is not a zero followed by more digits including zero.

- Zero or more occurrences is given by *
- We call the * operator Kleene closure, or closure for short.

Formalizing Regular Expressions

- A Regular Expression (RE) describes a set of strings over the characters contained in some alphabet, Σ, augmented with a character ε that represents the empty string.

- For a given RE, r, we denote the language that it specifies as $L(r)$

- A Regular Expression is built up from three basic operations

1. Alternation - The alternation, or union, of two sets of strings, R and S, denoted by $R \mid S$, is $\{ x \mid x \in R \text{ or } x \in S \}$

2. Concatenation - The concatenation of two sets R and S, denoted RS , contains all strings formed by prepending an element of R onto one from S, or $\{ xy \mid x \in R \text{ and } y \in S \}$

3. Closure - The Kleene closure of a set R, denoted R^* is

$$\bigcup_{i=0}^{\infty} R^i$$

The union of the concatenations of R with itself, zero or more times

FORMALIZING REGULAR EXPRESSIONS

- A language that can be described by a Regular Expression is called a regular language
- Regular Expressions over a given alphabet Σ are the smallest set of expressions that consists of :

R	$=$	ε			
	$\mid$	$'c'$	*where*	$c \in \Sigma$	: Base cases for the Re
	$\mid$	R *or* R			: Union of REs
	$\mid$	RR			: Concatenation of REs
	$\mid$	R^*			: Closure of REs

Formalizing Regular Expressions – Extended Notation

- Since the introduction of Kleene closure in the 1950s, many extensions have been added to regular expressions to enhance their ability to specify string patterns

- All of the following can be derived using the previous three basic rules

- One or more instances - The unary, postfix operator + represents the positive closure of a regular expression. E.g. $R+$ denotes one or more instances of R

- Zero or one instance - The unary postfix operator ? means "zero or one occurrence." E.g. $R?$ is equivalent to $R \mid \varepsilon$

- Character classes - A regular expression $a_1 \mid a_2 \mid a_3 \mid \ldots \mid a_n$ where the a_i's are each symbols of the alphabet, can be replaced by the shorthand $[a_1a_2a_3\ldots a_n]$. When $a_1, a_2, a_3, \ldots, a_n$ is a logical sequence it can be replaced by the shorthand $[a_1\text{-}a_n]$
E.g: a|b|c|…|z is equal to [a-z]

- Complement operator - The notation ^c specifies the set $(\Sigma - c)$ the complement of c with respect to Σ. In other words it represents "any character except the ones listed."
E.g. [^A-Za-z] matches any character that is not an uppercase or lowercase letter.

- Parenthesis – You can group parts of an RE with round brackets - ( ) or parentheses,
This allows to apply a quantifier to the entire group or to restrict alternation to part of the RE.

Regular Expressions – Some Examples

- Identifiers for C/C++/Java (Algol type languages) [_a-zA-Z][_a-zA-Z0-9]*
- identifiers limited to six characters [_a-zA-Z][_a-zA-Z0-9]{5}
- unsigned integers 0|[1-9][0-9]* , in practice many implementations accept [0-9]+
- signed integers [+-]?[0-9]+
- signed real numbers [-+]?[0-9]+\.[0-9]+ - optional sign, mandatory integer, and fraction
- Floating point numbers in scientific notation

  [-+][0-9]+\.[0-9]+[eE][-+]?[0-9]+ Mandatory sign, integer, fraction, and exponent

  [-+]?[0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)? Optional sign, mandatory integer, optional fraction and exponent

Formalizing Regular Expressions

- Recall that for a given RE, *r*, we denote the language that it specifies as *L(r)*
- But what does a language actually mean in this case ?

- A language defined by a regular expression is *all* the set of strings that can be described by that regular expression

Examples :

- *L(ε)* = { " " }

- *L('c')* = { "c" }

- *L(1*)* = { " ", "1", "11", "111", "1111", …… } = all strings of *1*s or the empty string

- *L([0-9]+)* = { "0", "1", "2", "3", "4", "5", "6", "7", "8", "9" , "10",11", …. }
  = unsigned integers

We say that "130" ∈ *L(* [0-9]+*)*
i.e. the string 130 belongs to
the language specified by the
regular expression [0-9]+

Example – Writing REs for the Lexemes of each Token Class

- Given what we learn about writing REs we can now write a specification for each of the token classes, say for a C/C++/Java type programming language

keyword	=	if|else|while|int|float| …				
digit	=	0|1|2|3|4|5|6|7|8|9			=	[0–9]
digits	=	digit digit*	=	digit+	=	[0–9]+
num	=	digits (fraction)? (exponent)?				
	=	digits(\.digits)? ([Ee][+-]?digits)?				
	=	digit+(\.digit+)? ([Ee][+-]?digit+)?				
	=	[0-9]+(\.[0-9]+)? ([Ee][+-]?[0-9]+)?			=	RE for floats and integers

identifier – strings of letters and digits starting with a letter or an underscore

= [_a–zA–Z][_a–zA–Z0–9]*

white space – non-empty sequence of blanks, tabs and new lines

= [\t\n]+

OpenParen = (

op = \+|\-| …

TOKENIZING

❑ Construct an RE matching all lexemes for all tokens - simply take the union of all the REs for all the token classes

R = RE(keyword) | RE(num) | RE(identifier) | RE(white space) | …

$R = R_1 \mid R_2 \mid R_3 \mid R_4 \mid \ldots$

❑ Now that we have a Regular Expression R, for matching all lexemes for all tokens, the steps taken by the Lexer to tokenize an input sequence of characters can be stated as follows :

1. Given an input sequence of characters $C_1, C_2, C_3, C_4 \ldots C_n$ to the Lexer we check whether some i ($1 \leq i \leq n$) number of characters belongs to the language of R,

   That is, for $1 \leq i \leq n$ check if
   $C_1, \ldots, C_i \in L(R)$

2. If true, then we know that $C_1, \ldots, C_i \in L(R_j)$ for some j (i.e. R_j is one of R_1 or R_2 or R_3 or R_4 or …)

3. Remove $C_1, \ldots, C_i$ from the input sequence and go to 1.

- What if different number of characters matches R ?

$C_1,\ldots,C_i \in L(R)$ for some i $(1 \le i \le n)$ number of characters AND
$C_1,\ldots,C_j \in L(R)$ for some j $(1 \le j \le n,\ j \ne i)$ number of characters

$$\underbrace{\underbrace{C_1, C_2, C_3, C_4, C_5, C_6, C_7, C_7, C_8, C_9, C_{10},\ldots\ldots\ldots,C_i}_{i},\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots C_j}_{j},\ldots\ldots\ldots\ldots$$

Solution : Always select (i.e. match) the longer sequence – maximal munch

- Which token should be used if more than one token matches ?

E.g. "for" can be both a keyword and an identifier

for some i $(1 \le i \le n)$ number of characters
$C_1,\ldots,C_i \in L(R_j)$
$C_1,\ldots,C_i \in L(R_k)$ where $j \ne k$

Solution : Uses the token class specification listed first (i.e. use R_j if $j < k$)
E.g. Usually keyword are listed before identifiers, thus "for" will be always matched as a keyword token

TOKENIZING

- What if there is no match ?

$$C_1,\ldots,C_i \notin L(R)$$

- This is an error and so we define another regular expression that specify strings not belonging to the language

error = All strings not belonging to the language specified by R

- error should have the least priority in our list of specifications for token classes

$$R = R_1 \mid R_2 \mid R_3 \mid R_4 \mid \ldots \mid R_{error}$$

Building a Scanner from Regular Expressions

- Regular expressions have become the basis for writing specifications for lexers, for example write a specification for a scanner generator such as Lex or Flex

$$
\begin{array}{rcl}
digit & \rightarrow & [0\text{-}9] \\
digits & \rightarrow & digit^{+} \\
number & \rightarrow & digits\ (.\ digits)?\ (\ \mathtt{E}\ [+-]?\ digits\ )? \\
letter & \rightarrow & [\mathtt{A\text{-}Za\text{-}z}] \\
id & \rightarrow & letter\ (\ letter \mid digit\ )^{*} \\
if & \rightarrow & \mathtt{if} \\
then & \rightarrow & \mathtt{then} \\
else & \rightarrow & \mathtt{else} \\
relop & \rightarrow & < \mid > \mid <= \mid >= \mid = \mid <>
\end{array}
$$

Regular definitions

LEXEMES	TOKEN NAME	ATTRIBUTE VALUE
Any *ws*	–	–
if	**if**	–
then	**then**	–
else	**else**	–
Any *id*	**id**	Pointer to table entry
Any *number*	**number**	Pointer to table entry
<	**relop**	LT
<=	**relop**	LE
=	**relop**	EQ
<>	**relop**	NE
>	**relop**	GT
>=	**relop**	GE

Tokens, their patterns, and attribute values

Building a Scanner from Regular Expressions

- Regular expressions have become the basis for writing specifications for lexers, for example write a specification for a scanner generator such as Lex or Flex -- Example portion of a Lex program

- The operators **{ }** specify either repetitions (if they enclose numbers) or definition expansion (if they enclose a name)

For identifiers (`id`) :
- **`int installID()`** – called to place the lexeme found in the symbol table
- **`yylval`** – pointer to the symbol table

- The token name `ID` is returned to the parser

For numbers (`number`) :
- **`int installNum()`** – puts numerical constants in to a separate table

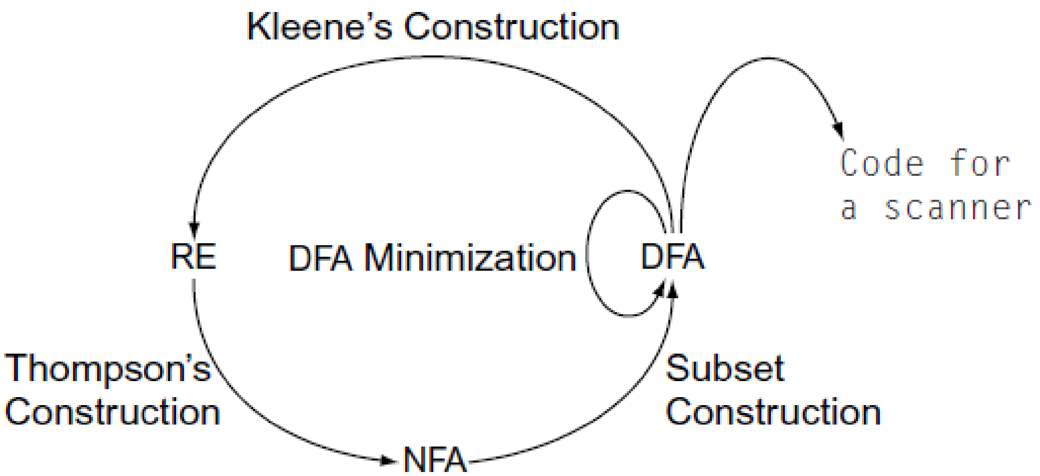
```
/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}
```

See Dragon book 2nd Ed. Sec 3.5 for more details

BUILDING A SCANNER FROM REGULAR EXPRESSIONS

- While lexical analyser generators (such as Lex and Flex) automate the creation of scanners, implementation of that software requires the simulation of a DFA – this is what we are going to learn next

- This section develops the constructions that transform an RE into an FA that is suitable for direct implementation



- Distinguish between Non-Deterministic FAs (NFA) and Deterministic FAs (DFA)
- Thompson's construction, derives an NFA from an RE
- The subset construction, builds a DFA that simulates an NFA
- Hopcroft's algorithm, minimizes a DFA
- Kleene's construction derives an RE from a DFA – but not a direct part of scanner implementation

The FA for new or not or while

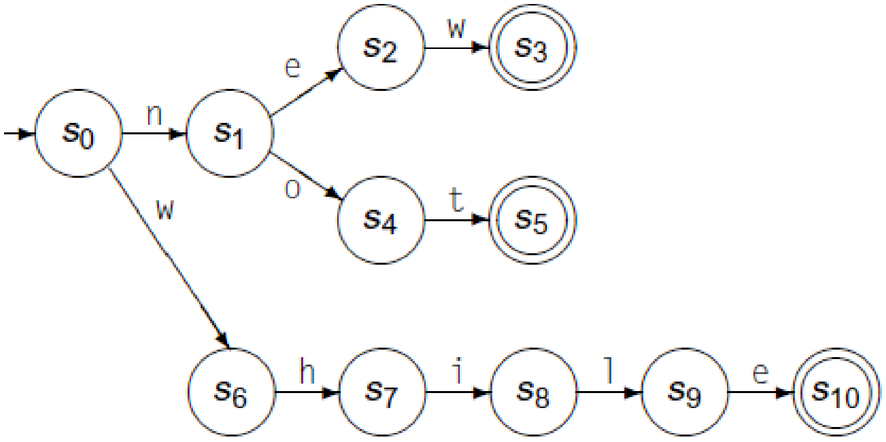
Set of states $S = \{s_0, s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8, s_9, s_{10}, s_e\}$

Alphabet $\Sigma = \{\mathtt{e}, \mathtt{h}, \mathtt{i}, \mathtt{l}, \mathtt{n}, \mathtt{o}, \mathtt{t}, \mathtt{w}\}$

Transition function
$$\delta = \left\{ \begin{array}{lllll} s_0 \xrightarrow{\mathtt{n}} s_1, & s_0 \xrightarrow{\mathtt{w}} s_6, & s_1 \xrightarrow{\mathtt{e}} s_2, & s_1 \xrightarrow{\mathtt{o}} s_4, & s_2 \xrightarrow{\mathtt{w}} s_3, \\ s_4 \xrightarrow{\mathtt{t}} s_5, & s_6 \xrightarrow{\mathtt{h}} s_7, & s_7 \xrightarrow{\mathtt{i}} s_8, & s_8 \xrightarrow{\mathtt{l}} s_9, & s_9 \xrightarrow{\mathtt{e}} s_{10} \end{array} \right\}$$

Start state $s_0 = s_0$

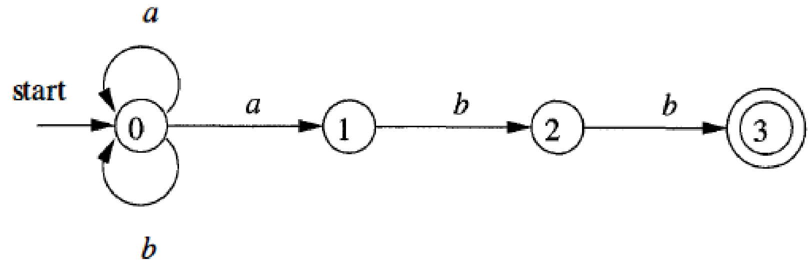
Accepting States $S_A = \{s_3, s_5, s_{10}\}$



NONDETERMINISTIC FINITE AUTOMATA

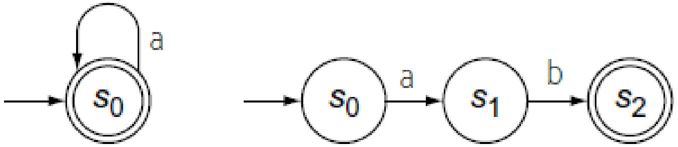
- A Nondeterministic Finite Automaton is an FA that allows transitions on the empty string, ε, and states that have multiple transitions on the same character

- Consider the transition graph for an FA recognizing the language of regular expression (a|b)*abb

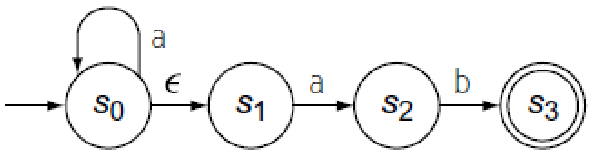


- The FA is non-deterministic as there is multiple edges labelled a out of state 0

- Now consider the FAs for the REs *a** and *ab*



- We can combine them with an ε -transition (i.e. empty string) to form an FA for a*ab

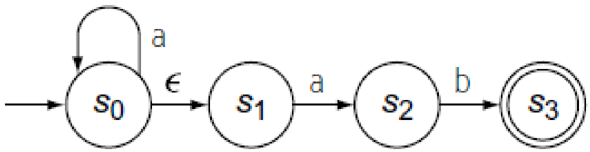


- Again we get non-determinism as the ε - transition, in effect, gives the FA two distinct transitions out of S_0 on the letter a

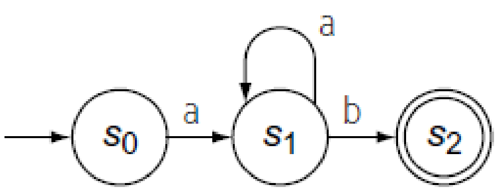
- In contrast a Deterministic Finite Automaton (DFA) is an FA where the states have only a single transition on the same character and does not have any ε transitions.

- Essentially a DFA is a special case of an NFA.
- Any NFA can be simulated by a DFA — which we will see how later

- But for a simple example – consider the following NFA :



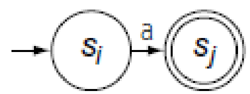
- The DFA that simulates the NFA can be written as :



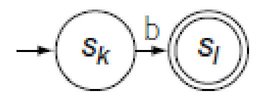
- Note how, the DFA does not have multiple transitions labelled by the same character out of any state

Converting an RE into an NFA – Thompson's Construction

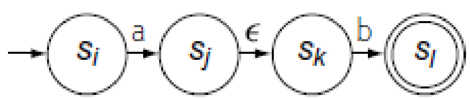
- We essentially use a template for building an NFA that corresponds to
  - A single-letter RE and
  - A transformation on NFAs that models the effect of each basic RE operator: concatenation, alternation, and closure
  - Apply the transformations in the order dictated by precedence and parentheses



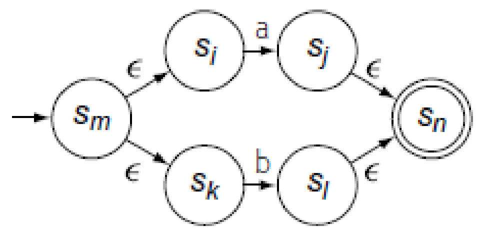
(a) NFA for "*a*"



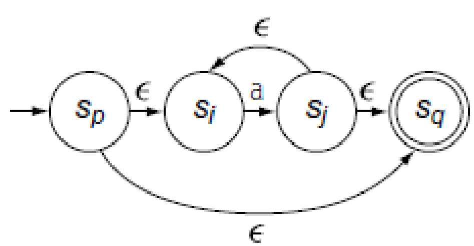
(b) NFA for "*b*"



(c) NFA for "*ab*"



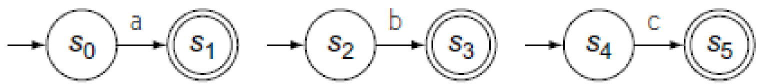
(d) NFA for "*a* | *b*"



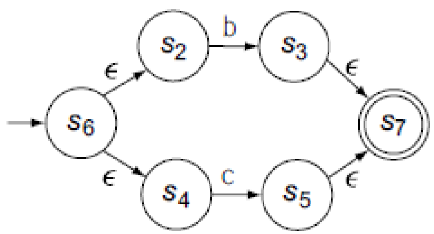
(e) NFA for "*a**"

Converting an RE into an NFA – Examples

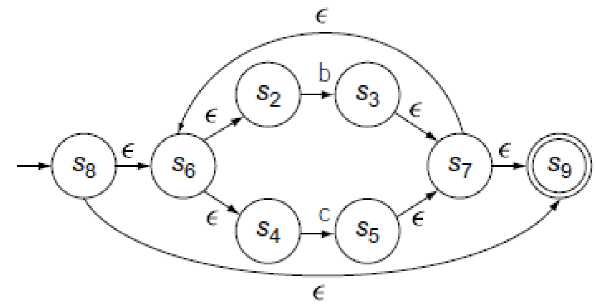
- Applying Thompson's Construction to a(b|c)*



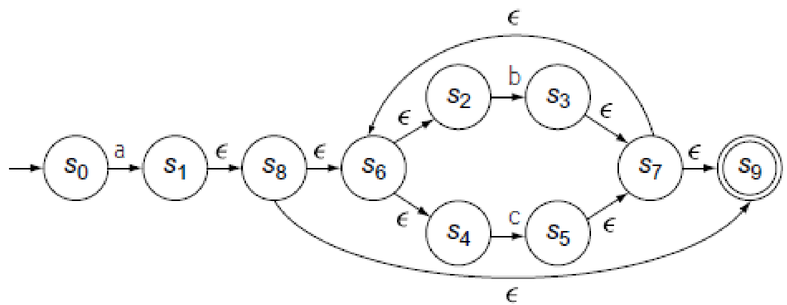
(a) NFAs for "*a*", "*b*", and "*c*"



(b) NFA for "*b* | *c*"



(c) NFA for "(*b* | *c*)*"



(d) NFA for "*a*(*b* | *c*)*"

- First build NFAs for a, b and c
- Parenthesis have higher precedence

Build NFA for b|c

- Closure has higher precedence than concatenation

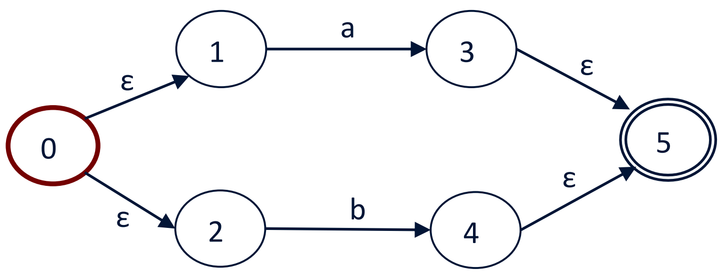
Build (b|c)*

- Finally concatenate a to (b|c)*

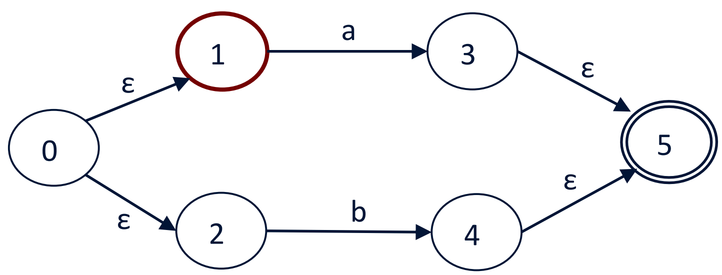
NFA to DFA: The Subset Construction

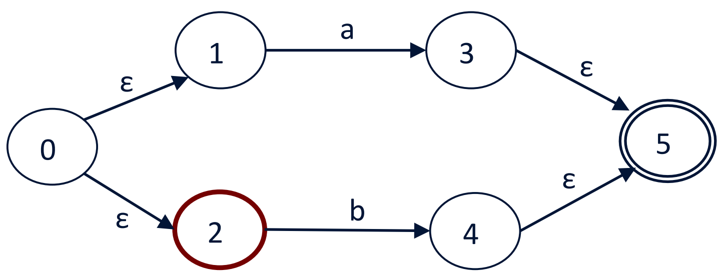
- NFA often has choice of making a transition on ε or on a real input symbol – thus simulation of an NFA is less straightforward than for a DFA
- Therefore we need to convert an NFA into a DFA for efficient implementation

- Consider the following NFA, where the alphabet is {a,b} and assume that we are at the start state (0) :



- There is a nondeterministic choice of ε-transitions, assume we take both choices simultaneously:

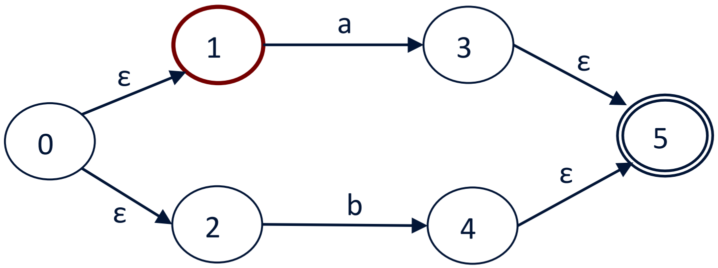


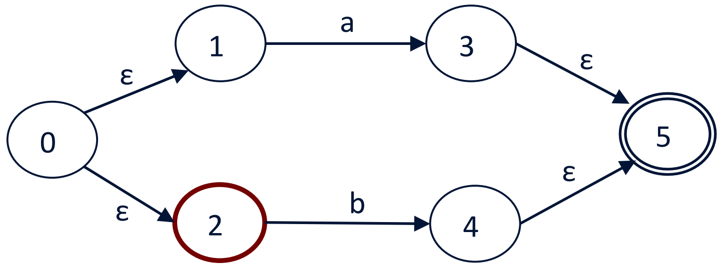


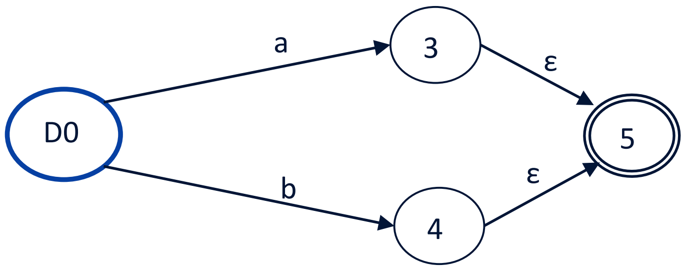
- Observe that we do not consume any input symbol to get into these two <u>configurations</u> of the NFA – in fact all of states 0,1 and 2 can be thought of as <u>one combined state</u>

NFA to DFA: The Subset Construction

- All of states 0,1 and 2 can be thought of as one combined state







- Combining NFA states based on ε-transitions allows us to eliminate ε-transitions when constructing the DFA

- So in essence what the subset construction algorithm does is combine NFA states that can be reached through ε-transitions to a single "subset" of states and only consider transitions based on the input symbols between different subsets

- The set of NFA states that can be reached from some NFA state n along paths containing only ε-transitions is called ε-*closure(n)*

NFA to DFA: The Subset Construction

When we have several choices of a next state in the NFA, we take all of the choices simultaneously and form a set of the possible next-states - such a set of NFA states will become a single DFA state

Algorithm : Take an NFA $(N, \Sigma, \delta_N, n_0, N_A)$ and convert it into a DFA $(D, \Sigma, \delta_D, d_0, D_A)$

Step 1: Start state of D (i.e. d_0) consists of n_0 (i.e. The start state of the NFA) and any states that can be reached from n_0 along paths containing only ε-transitions. We call this set of states ε-$closure(n_0)$
Step 2 : Add d_0 to the list of DFA states called *Dstates*
Step 3 : We say that d_0 is "unmarked" and enter into the following while loop:

```
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a ) {
        U = ϵ-closure(move(T, a));
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}
```

U is the DFA state that represent the NFA states returned by the *ε-closure(move(T,a))*

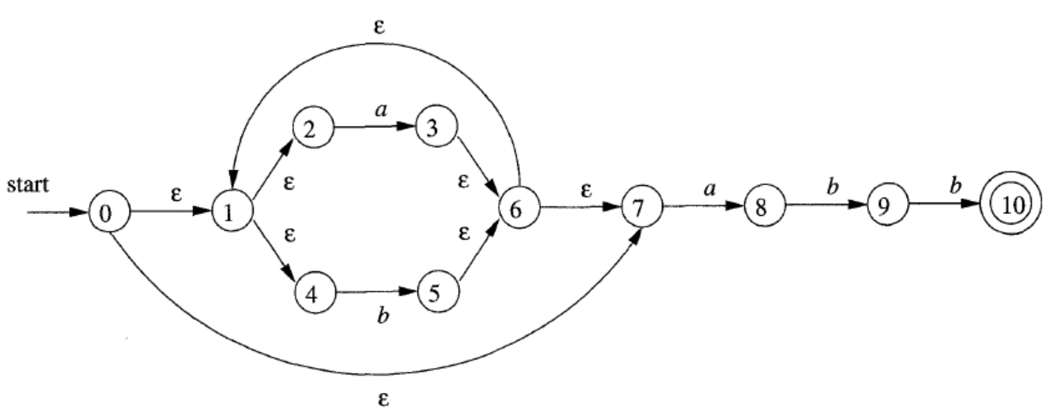
Set of NFA states that can be reached from *T* given an input symbol of *a* and *ε*-transitions

Add to the DFA's Transition function *Dtran*, indicating that you can reach *U* from *T* on receiving an input symbol of *a*

NFA TO DFA: THE SUBSET CONSTRUCTION - EXAMPLE

- Lets convert the NFA for *(a|b)*abb* to a DFA using the subset construction algorithm
- ε*-closure*(n_0) = ε*-closure*(0) = {0, 1, 2,4, 7} we call this DFA state *A*, add *A* to *Dstates*
- Now go into the while loop, with *Dstates* containing only one unmarked state, i.e. *A*

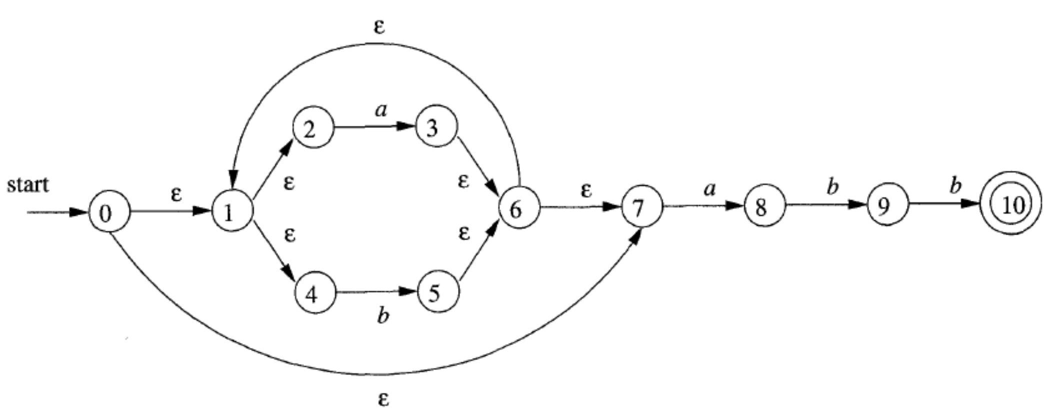
```
while ( there is an unmarked state T in Dstates ) {
        mark T;
        for ( each input symbol a ) {
                U = ε-closure(move(T, a));
                if ( U is not in Dstates )
                        add U as an unmarked state to Dstates;
                Dtran[T, a] = U;
        }
}
```



Example : Convert the NFA for (a|b)*abb to a DFA

- The input alphabet is *{a, b}*

```
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a ) {
        U = ε-closure(move(T, a));
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}
```



mark *A*
compute *Dtran[A, a] = ε-closure(move(A,a))*

move(A,a) = { 3, 8 }
ε-closure(move(A,a)) = {1,2,3,4,6,7,8}

Let the set of NFA states *{ 1,2,3,4,6,7,8 }* be represented by the DFA state *B*

Similarly,
Compute *Dtran[A, b] = ε-closure(move(A,b))*
Dtran[A, b] = {1,2,4,5,6,7} → DFA state *C*

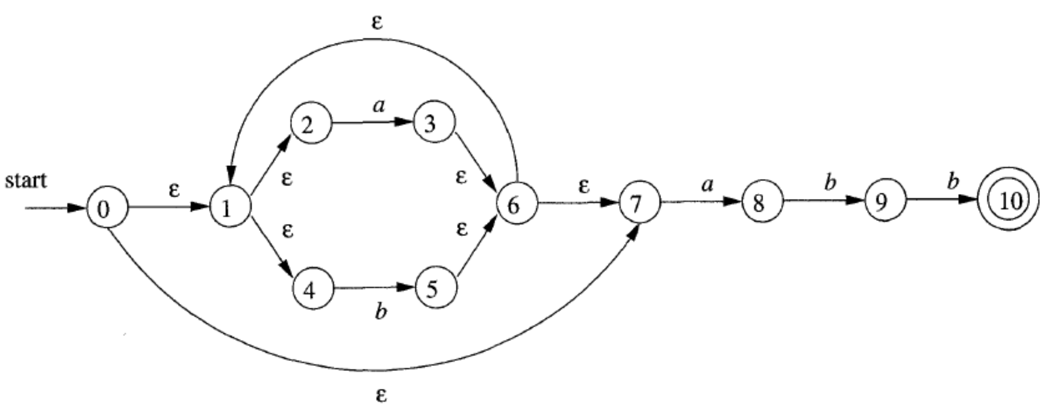
EXAMPLE : CONVERT THE NFA FOR (a|b)*abb TO A DFA

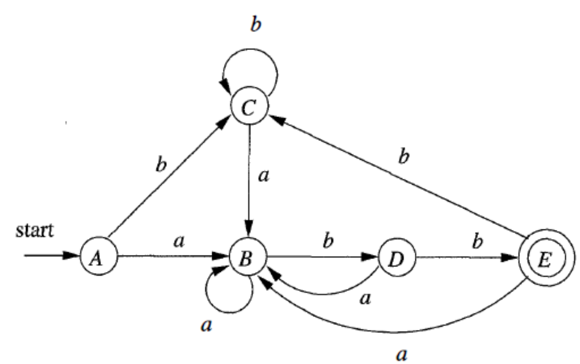
- Continuing this process with the unmarked sets *B* and *C*, we eventually reach a point where all the states of the DFA are marked
- At this point we will have the following transition table

```
while ( there is an unmarked state T in Dstates ) {
    mark T;
    for ( each input symbol a ) {
        U = ε-closure(move(T, a));
        if ( U is not in Dstates )
            add U as an unmarked state to Dstates;
        Dtran[T, a] = U;
    }
}
```

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

This is basically the DFA that simulates the NFA for *(a|b)*abb*





Number of DFA States

- Given an NFA with N states it can be proven that there are $2^N - 1$ non-empty sets of NFA states

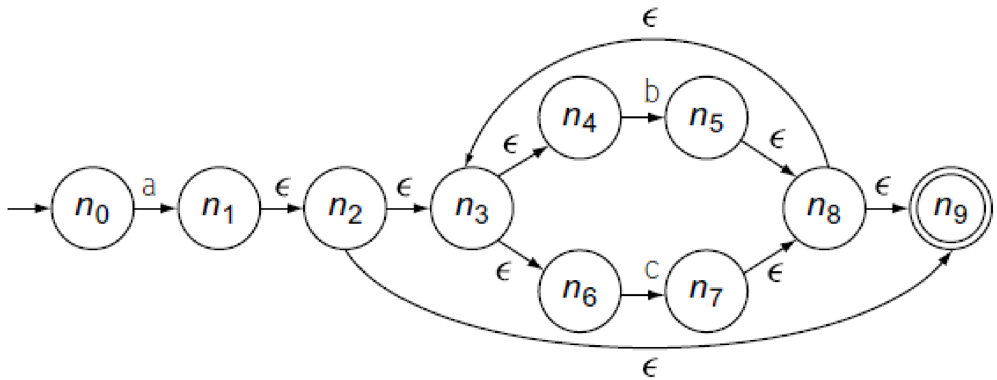
- This implies that a DFA that will simulate the NFA could potentially have exponentially more number of states than the NFA !

- However the set of states in the DFA is finite and the DFA still makes one transition per input symbol.

- Thus, the DFA that simulates the NFA still runs in time proportional to the length of the input string

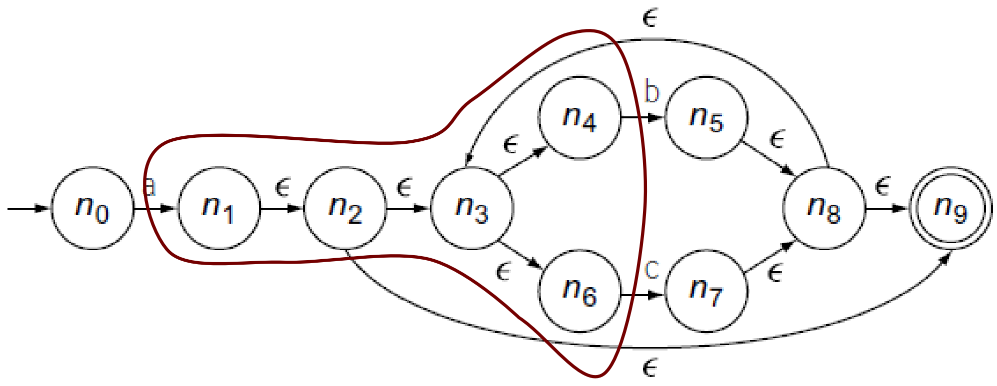
- The simulation of an NFA on a DFA has a potential space problem, but not a time problem

Another Example : Convert the NFA for a(b|c)* to a DFA



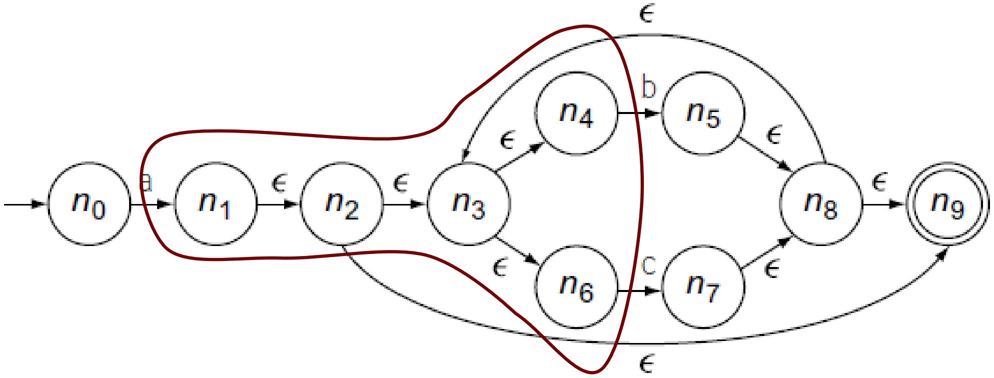
NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	n1,n2,n3,n4,n6,n9	-	-

Another Example : Convert the NFA for a(b|c)* to a DFA



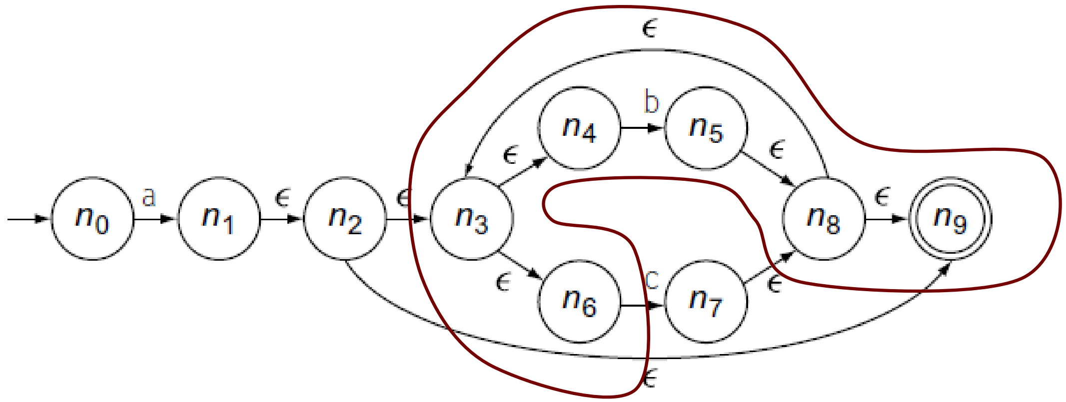
NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	n1,n2,n3,n4,n6,n9	-	-
n1,n2,n3,n4,n6,n9	d1			

Another Example : Convert the NFA for a(b|c)* to a DFA



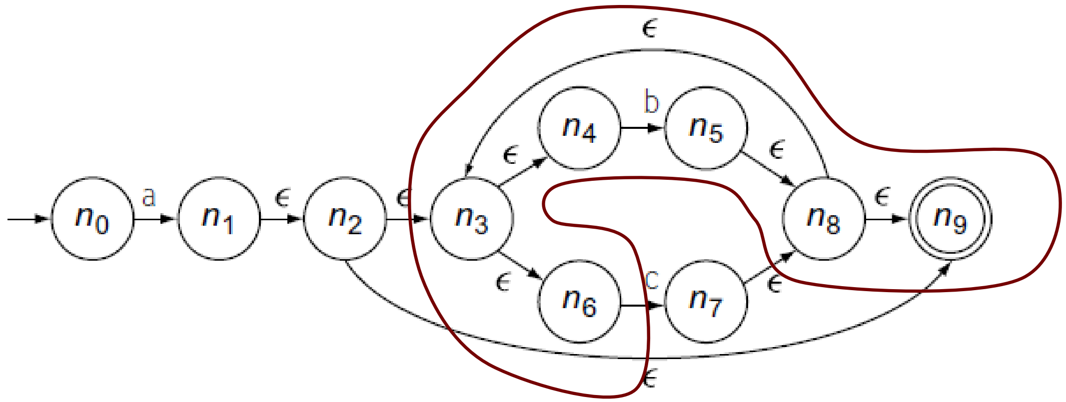
NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	n1,n2,n3,n4,n6,n9	-	-
n1,n2,n3,n4,n6,n9	d1	-	n5,n8,n9,n3,n4,n6	n7,n8,n9,n3,n4,n6

ANOTHER EXAMPLE : CONVERT THE NFA FOR a(b|c)* TO A DFA



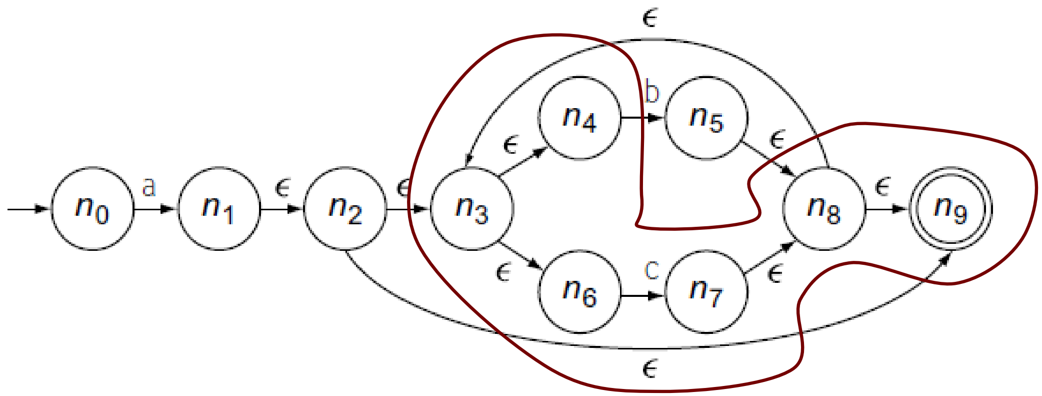
NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	n1,n2,n3,n4,n6,n9	-	-
n1,n2,n3,n4,n6,n9	d1	-	n5,n8,n9,n3,n4,n6	n7,n8,n9,n3,n4,n6
n5,n8,n9,n3,n4,n6	d2			
n7,n8,n9,n3,n4,n6	d3			

Another Example : Convert the NFA for a(b|c)* to a DFA



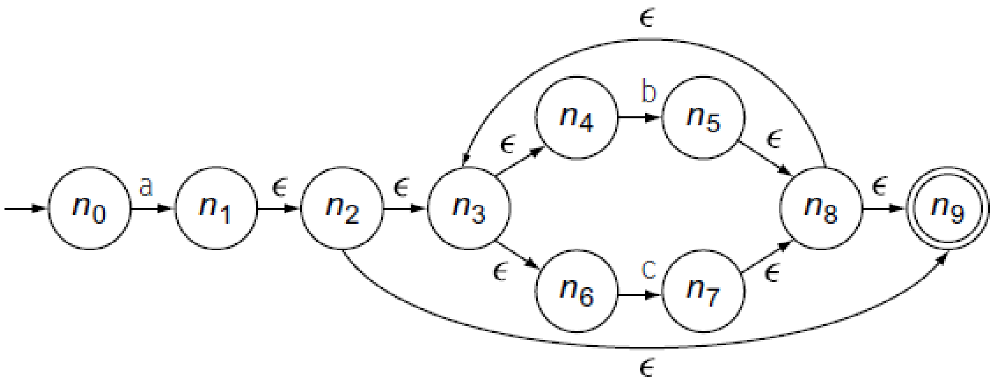
NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	n1,n2,n3,n4,n6,n9	-	-
n1,n2,n3,n4,n6,n9	d1	-	n5,n8,n9,n3,n4,n6	n7,n8,n9,n3,n4,n6
n5,n8,n9,n3,n4,n6	d2	-	-	n7,n8,n9,n3,n4,n6
n7,n8,n9,n3,n4,n6	d3			

Another Example : Convert the NFA for a(b|c)* to a DFA

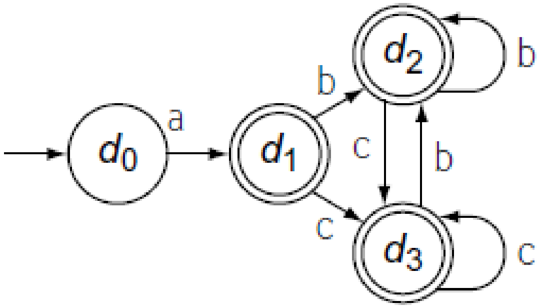


NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	n1,n2,n3,n4,n6,n9	-	-
n1,n2,n3,n4,n6,n9	d1	-	n5,n8,n9,n3,n4,n6	n7,n8,n9,n3,n4,n6
n5,n8,n9,n3,n4,n6	d2	-	d2	n7,n8,n9,n3,n4,n6
n7,n8,n9,n3,n4,n6	d3	-	n5,n8,n9,n3,n4,n6	d3

ANOTHER EXAMPLE : CONVERT THE NFA FOR a(b|c)* TO A DFA

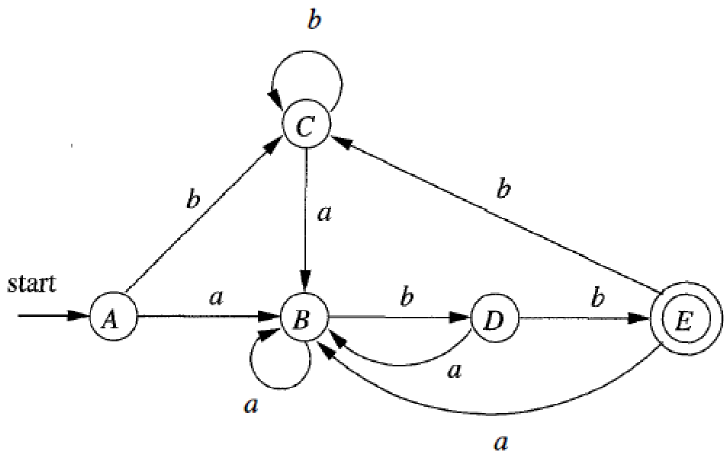


NFA states	DFA state	*ε-closure*(move(T,a))	*ε-closure*(move(T,b))	*ε-closure*(move(T,c))
n0	d0	d1	-	-
n1,n2,n3,n4,n6,n9	d1	-	d2	d3
n5,n8,n9,n3,n4,n6	d2	-	d2	d3
n7,n8,n9,n3,n4,n6	d3	-	d2	d3



DFA to Minimal DFA: Hopcroft's Algorithm

- The DFA that emerges from the subset construction can have a large number of states
- Some states can be merged : e.g. in the previous DFA *A* and *C* have the same move function



- To minimize the number of states in a DFA we need a technique to detect when two states are equivalent—that is, when they produce the same behaviour on any input string

DFA to Minimal DFA: Hopcroft's Algorithm

- The algorithm works by partitioning the states of a DFA into groups of states that cannot be distinguished – i.e. produce the same behaviour on any input string
- Each group of states is then merged into a single state of the new minized DFA

Algorithm

Step 1: Given D states in the DFA partition the states into two groups – F and $S\text{-}F$, the accepting states and the nonaccepting states. Denote this initial partitioning as $P = F, S\text{-}F$

Step 2: Let new partitioning $P_{new} = P$

For (Each group G in P_{new}) {

partition into subgroups such that two states s and t are in the same subgroup if and only if for all input symbols a, states s and t have transitions on a to states in the same group G

/* worst case, a state will be in a subgroup by itself * /

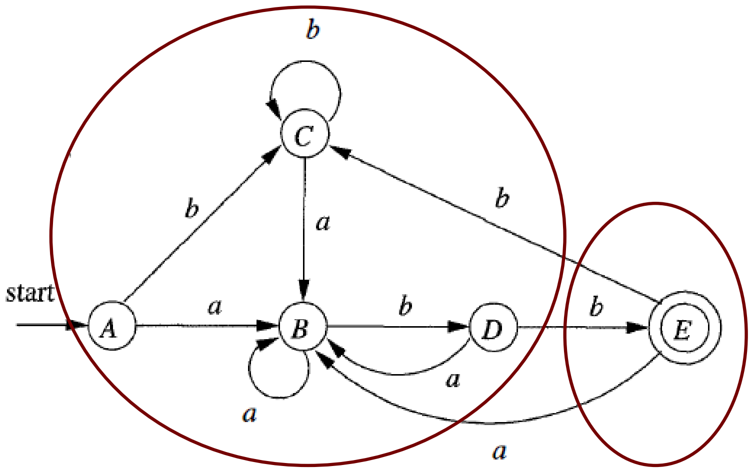
replace G in P_{new} by the set of all subgroups formed

}

Step 3: If there is no change in P_{new}, i.e. $P_{new} = P$ then let $P_{final} = P$, go to Step 4. Else go to Step 2 and repeat with P_{new} in place of P

Step 4: Choose one state in each group of P_{final} as the representative for that group. The representatives will be the states of the minimum-state DFA D'.

DFA to Minimal DFA: Hopcroft's Algorithm - Example



- Initial partitioning P = {A, B, C, D} { E }
  i.e. the accepting and nonaccepting states

- Now consider both groups {A, B, C, D} and { E }
  and inputs a and b

- No further split possible for { E }, but can consider splitting the group {A, B, C, D}

- On an input symbol of a all states goes to states within the same group
- On an input symbol of b states A, B, and C go to members of group {A, B, C, D}, while state D goes to E, a member of another group

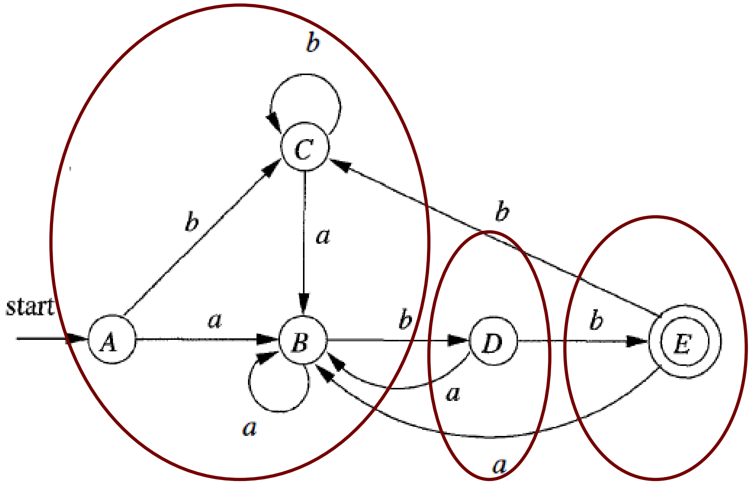
- Thus we split group {A, B, C, D} into {A, B, C} {D} and P_{new} for this round is {A, B, C} {D} {E}

- Set P = P_{new} and repeat

DFA TO MINIMAL DFA: HOPCROFT'S ALGORITHM - EXAMPLE

- P = {A, B, C} {D} {E}
- Can split {A, B, C} into {A, C} {B} , since A and C each go to a member of {A, B, C} on input b, while B goes to a member of another group, {D}

- P_{new} for this round is {A, C} {B} {D} {E}
- Set P = P_{new} and repeat

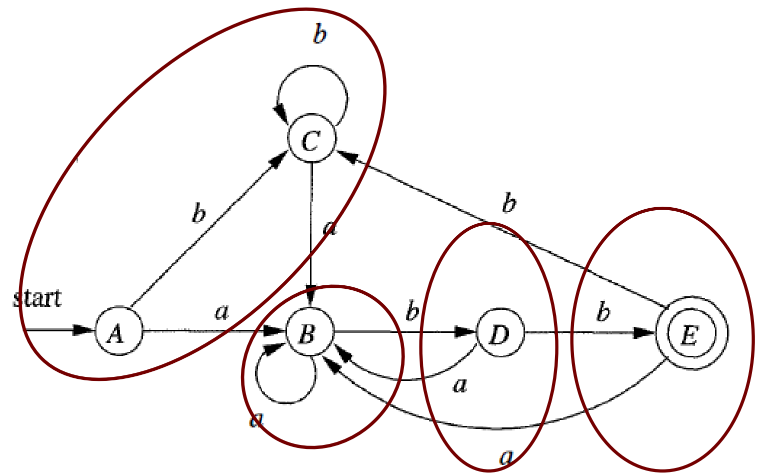


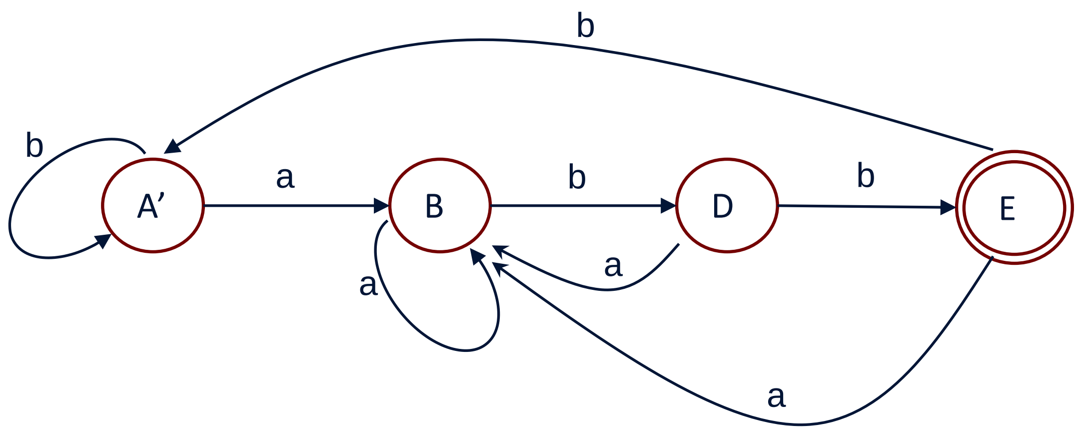
DFA to Minimal DFA: Hopcroft's Algorithm - Example

- ❑ P = {A, C} {B} {D} {E}

- ❑ But we cannot split the one remaining group with more than one state, since A and C each go to the same state (and therefore to the same group) on each input.

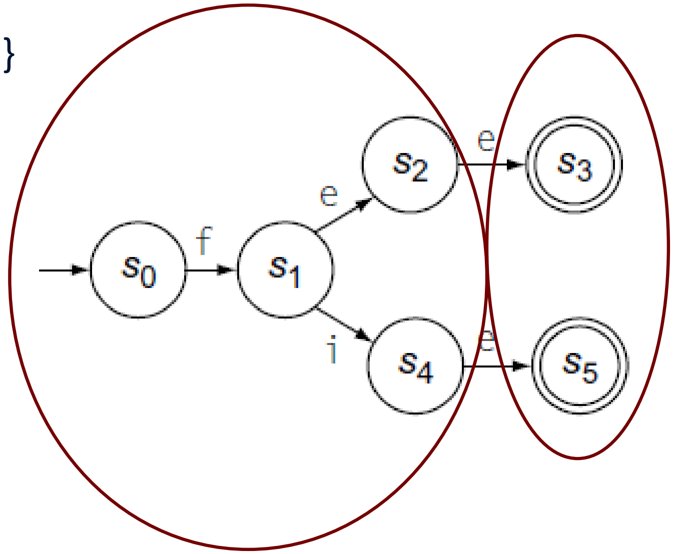
- ❑ Thus P_{final} = {A, C} {B} {D} {E}





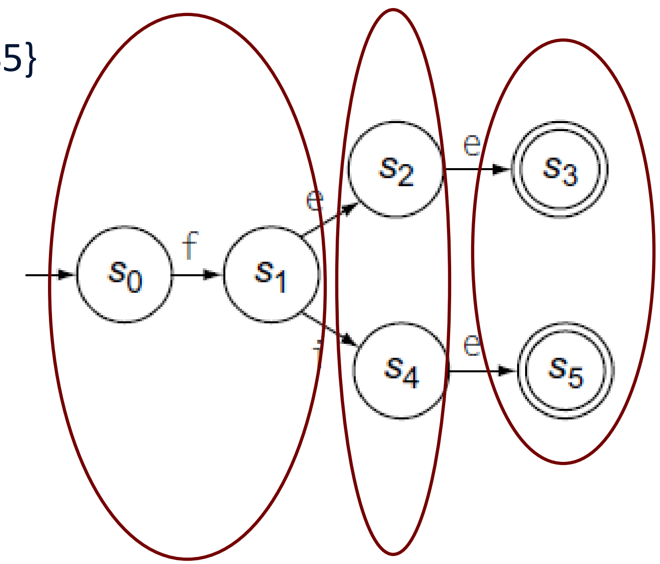
DFA to Minimal DFA: Another Example – Very quickly !!

- DFA for *fee* | *fie*
- P = {S0,S1,S2,S4} {S3,S5}
- P_{new} = {S0,S1,S2,S4} {S3,S5}



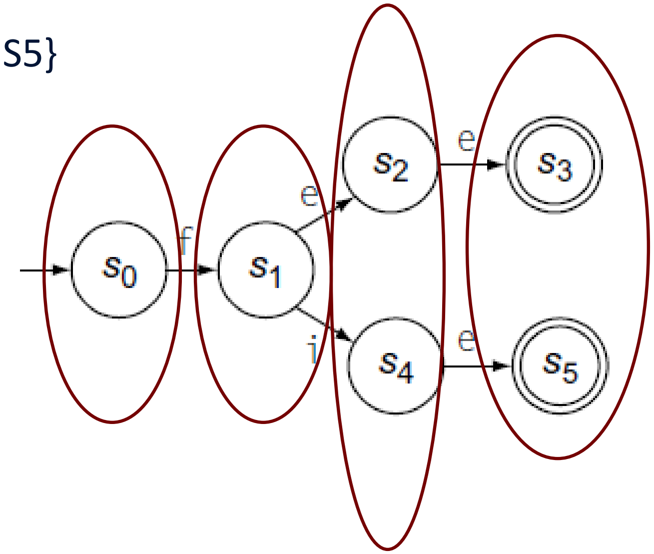
DFA to Minimal DFA: Another Example – Very quickly !!

- DFA for *fee* | *fie*
- P = {S0,S1,S2,S4} {S3,S5}
- P_{new} = {S0,S1} {S2,S4} {S3,S5}



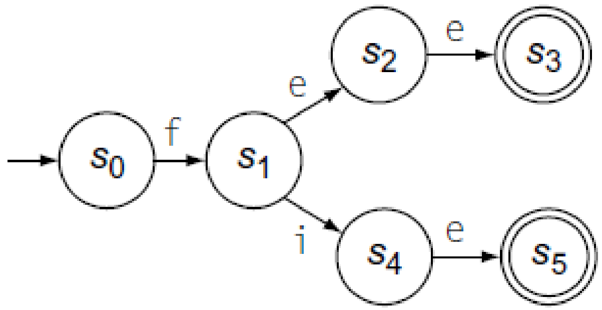
DFA to Minimal DFA: Another Example – Very quickly !!

- DFA for *fee* | *fie*
- P = {S0,S1} {S2,S4} {S3,S5}
- P_{new} = {S0} {S1} {S2,S4} {S3,S5}

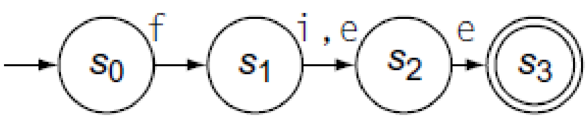


DFA to Minimal DFA: Another Example – Very quickly !!

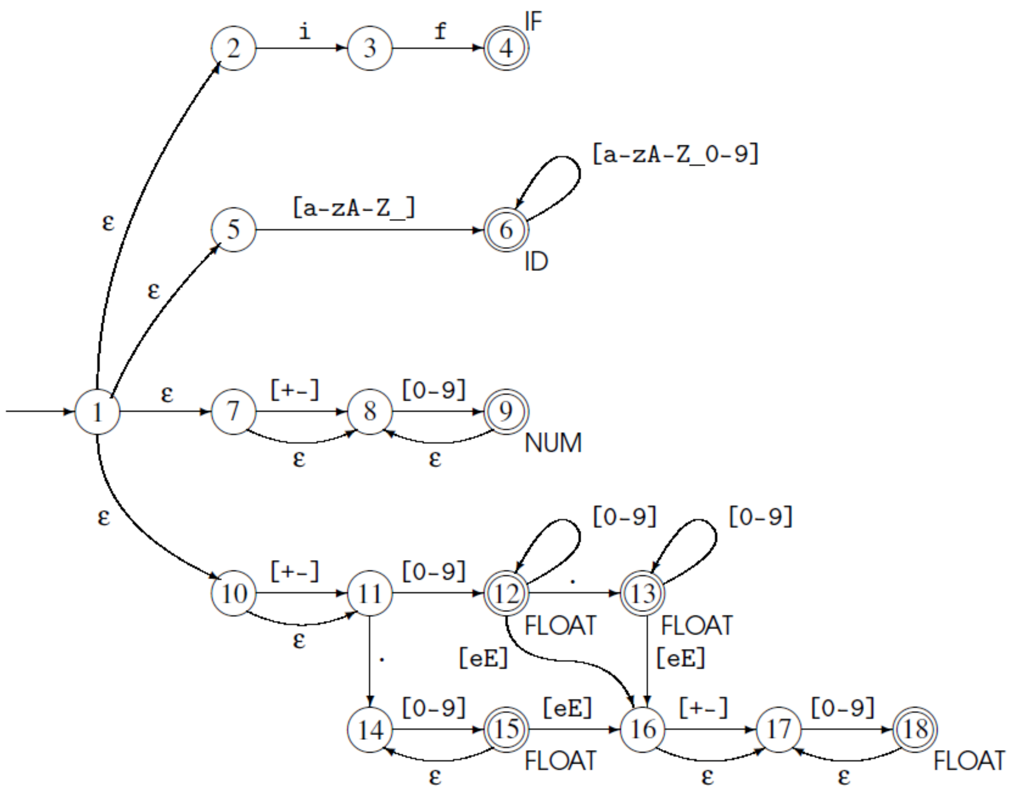
- DFA for *fee* | *fie*



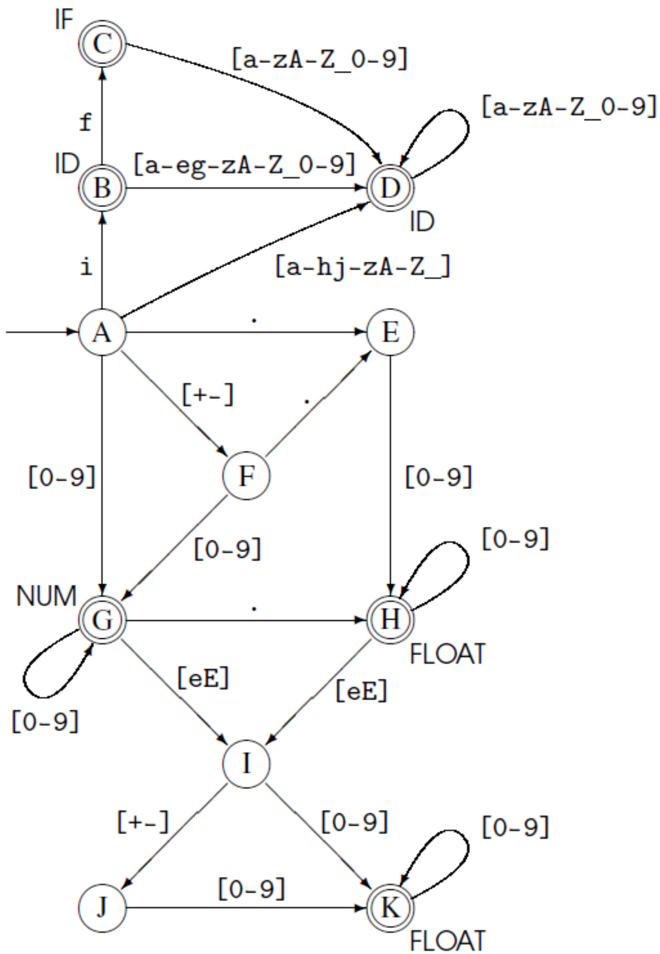
Step	Current Partition	Examines		
		Set	Char	Action
0	$\{\{s_3,s_5\},\{s_0,s_1,s_2,s_4\}\}$	—	—	—
1	$\{\{s_3,s_5\},\{s_0,s_1,s_2,s_4\}\}$	$\{s_3,s_5\}$	*all*	*none*
2	$\{\{s_3,s_5\},\{s_0,s_1,s_2,s_4\}\}$	$\{s_0,s_1,s_2,s_4\}$	e	*split* $\{s_2,s_4\}$
3	$\{\{s_3,s_5\},\{s_0,s_1\},\{s_2,s_4\}\}$	$\{s_0,s_1\}$	i , e	*split* $\{s_1\}$
4	$\{\{s_3,s_5\},\{s_0\},\{s_1\},\{s_2,s_4\}\}$	*all*	*all*	*none*



COMBINED NFA AND THE RESULTING DFA FOR SEVERAL TOKENS



Combined NFA for “IF”, IDs, NUM and FLOAT



Combined DFA for “IF”, IDs, NUM and FLOAT

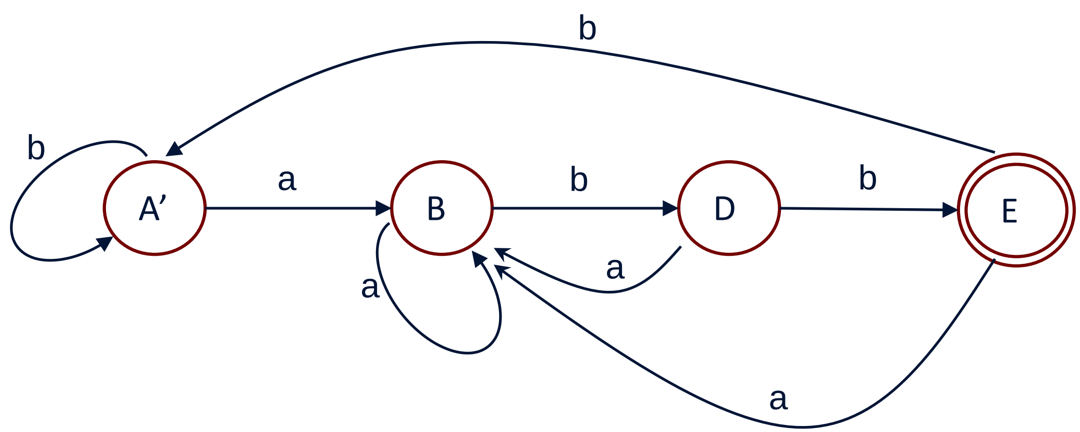
Implementing Scanners

- All the formalisms and algorithms we have learnt allows us to automate the construction of the scanner
  - The compiler writer creates an RE for each syntactic category
  - Gives the REs as input to a scanner generator (e.g. Lex or Flex)
  - Scanner generator builds NFA, DFA, minimal DFA

- At this point the scanner generator must convert the DFA into executable code
- The strategies are
  - Table driven scanner
  - Direct coded scanner
  - Hand-coded scanner

- We are not going to go into too much of details for these at this point … but one of the recommended text books does:
  - Cooper and Troczon, Engineering a Compiler - Section 2.5

1. Codify the DFA transitions in a table



	a	b	other
A’	B	A’	error
B	B	D	error
D	B	E	error
E	B	A’	error

2. Use the table to drive a skeleton scanner programme

δ	a	b	other
A’	B	A’	error
B	B	D	error
D	B	E	error
E	B	A’	error

Thomas Reps. *‘Maximal-munch’ tokenization in linear time*. ACM Transactions on Programming Languages and Systems, 20(2):259–273, March 1998
[See module online material for paper]

❑ Table driven scanner’s table lookups can be eliminated by <u>generating</u> a specialized code fragment to implement each state – this results in what we call a direct-coded scanner

```
procedure Tokenize(M: DFA, input: string)
let ⟨Q, Σ, δ, q0, F⟩ = M in
begin
  i := 1
  loop
    q := q0
    push(⟨Bottom, i⟩)
    /* Scan for tokens */
    while      i ≤ length(input)
           and δ(q, input[i]) is defined
    do
      if q ∈ F then reset the stack to empty fi
      push(⟨q, i⟩)
      q := δ(q, input[i])
      i := i + 1
    od
    /* Backtrack to the most recent final state */
    while q ∉ F do
      ⟨q, i⟩ := pop()
      if q = Bottom then
        return "Failure: tokenization not possible"
      fi
    od
    print(i − 1)
    if i > length(input) then
      return "Success"
    fi
  pool
end
```

Print the final character of a valid token

Further Reading

- Cooper and Troczon, Engineering a Compiler - Chapter 2
- Aho, Lam, Sethi, Ulman, Compilers Principals Techniques and Tools – Chapter 3
- Torben Mogensen, Basic Compiler Design – Chapter 2
- Thomas Reps. *'Maximal-munch' tokenization in linear time*. ACM Transactions on Programming Languages and Systems, 20(2):259–273, March 1998 -- See module webpage for paper