

Lexical Analysis:

Characters to Tokens

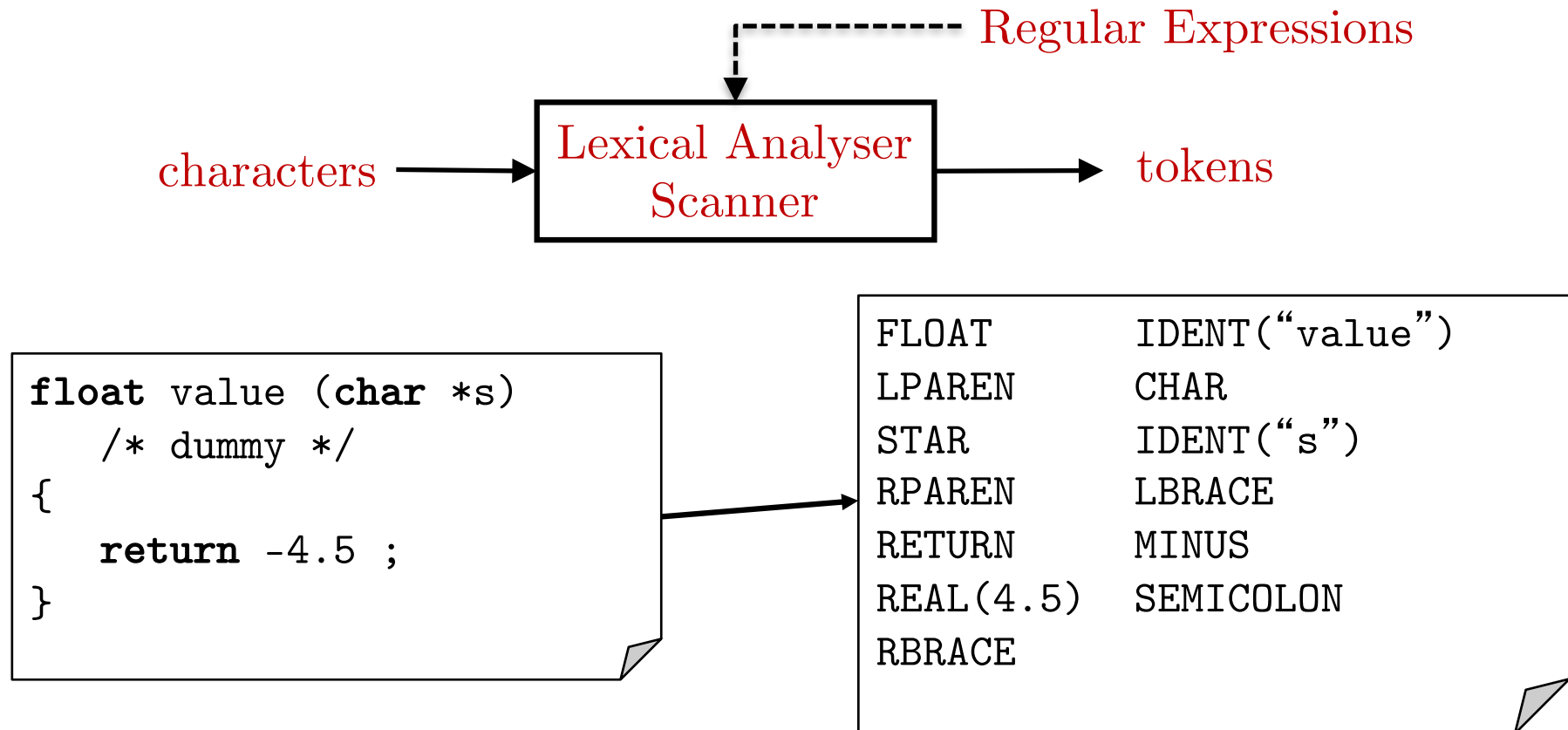
Naranker Dulay

n.dulay@imperial.ac.uk

<https://www.doc.ic.ac.uk/~nd/compilers>

Main idea

Lexical analysis transforms a stream of characters into a stream of tokens, ideally based on a *formal description* of the tokens of the input language



Identifier Tokens

Identifiers (names) are usually classified into:

- **Keyword identifiers** e.g. `class`, `package`, `while`. These have special meaning in a programming language and are normally represented by their own token e.g. `CLASS`, `PACKAGE`, `WHILE`. Small finite number in most languages.
- **Non-keyword identifiers** e.g. `year`, `quad09`. These are defined by the programmer and normally represented by a general identifier token, e.g. `IDENT` plus a string attribute holding the actual name. For example, `year` would be mapped to the token `IDENT("year")`

A scanner needs to be able to quickly determine if a scanned identifier is a keyword. **Use a fast string lookup function, e.g. a perfect hash function**

Literal Tokens

Literals (constants) are usually classified into:

- **Unsigned integers.** Represented by a literal token for integers, plus an integer attribute holding the actual value e.g. `INTEGER(123)`
What about negative integers? Big integers? Integer overflow?
- **Unsigned reals.** Represented by a literal token for reals, plus a floating-point attribute holding the actual value e.g. `REAL(123.78)`
- **Strings.** Represented by a literal token for strings, plus a string attribute holding the actual value e.g. `STRING("hello world")`

We need to take care that the compiler's language correctly represents values in the compiled language, e.g. Unicode strings in a compiler that only supports ASCII strings.

Other Tokens

- **1-char or 2-char symbols** e.g. + , <= (are normally represented by their own token i.e. PLUS, COMMA, LESSEQUAL, LPAREN
- **Whitespace** e.g. spaces, tabs, newlines are not normally represented in the token stream (unless they occur in a string literal). Whitespace is normally needed to separate adjacent identifiers, literals e.g. in **public static void**
- **Comments** are not normally represented in the token stream.
- **Preprocessing directives and macros** are normally removed in a stage before lexical analysis.

What data type(s) would you use for tokens?

What extra token information would be useful for reporting errors?

Regular Expressions (Regex)

Allow us to formally define the acceptable lexical tokens of the programming language – the strings of characters that match the regular expression

Regular Expression Matches

a	Symbol: a character (of the language's alphabet)
ϵ	Epsilon: the empty string
R1 R2	Concatenation: Regex R1 followed by regex R2
R1 R2	Alternation: Either regex R1 or regex R2
R*	Repetition: 0 or more occurrences of R.
(R)	Grouping: R itself.
\a	Regular-expression meta-char like * () \ etc

Precedence

Repetition (highest), concatenation, alternation (lowest)
Use parenthesis (grouping) to override

Regular Expression Shortcuts

The following are convenient shortcuts, they can be written with the core regular expressions (previous slide)

Shortcut

Matches

R?

Optional: 0 or 1 occurrence of R

R+

1 or more occurrences of R

[a-zA-Z]

CharSet: any character from the given set

[^a-zA-Z]

CharSet: any character **except** those in given set

•

AnyChar: any character **except** newline

Examples

Regular Expression

Matches

$(0|1)^+$

?

$(0|1)^*0$

?

$[0-9]^+$

?

$(+|-)^? [0-9]^+$

?

$[a-zA-Z][a-zA-Z0-9]^*$

?

$ba(na)^*$

?

Exercise

On Linux how would you write a command to list all hidden files and directories (those starting with a dot character) in your home directory?

Exercise

On Linux how would you write a command to copy all hidden files and directories (dot files/directories) in your home directory to a directory called DOT?

DO NOT TRY TESTING YOUR COMMAND !!!!

Regular Expression Rules

Rules (**productions**) of the form $\alpha \rightarrow X$ where α is a **non-terminal** (name of the rule) and X is a regular expression constructed from both **non-terminals** (names of rules) and **terminals** (input characters). Non-terminals must be defined before use in a rule. **Note: Recursion is not allowed in regular expression rules**

Examples

Digit	\rightarrow	[0-9]	
Int	\rightarrow	Digit +	
Signedint	\rightarrow	(+ -)? Int	
Keyword	\rightarrow	if while do	
Letter	\rightarrow	[a-zA-Z]	
Identifier	\rightarrow	Letter (Letter Digit)*	
Whitespace	\rightarrow	[\ \t \n]+	# escape space tab & newline
Comment	\rightarrow	#.*	# to end of line
Error	\rightarrow	.	# otherwise error

Disambiguation Rules

What if a sequence of characters can be matched by more than one regular expression?

→ Use **longest matching character sequence**

→ Otherwise assume regular expressions have **textual precedence** such that the earliest regular expression takes precedence, e.g. we would write the regex for keywords before the regex for identifiers.

Example

doughnut would match **Identifier** by the longest match rule

do would match **Keyword** by the textual precedence rule

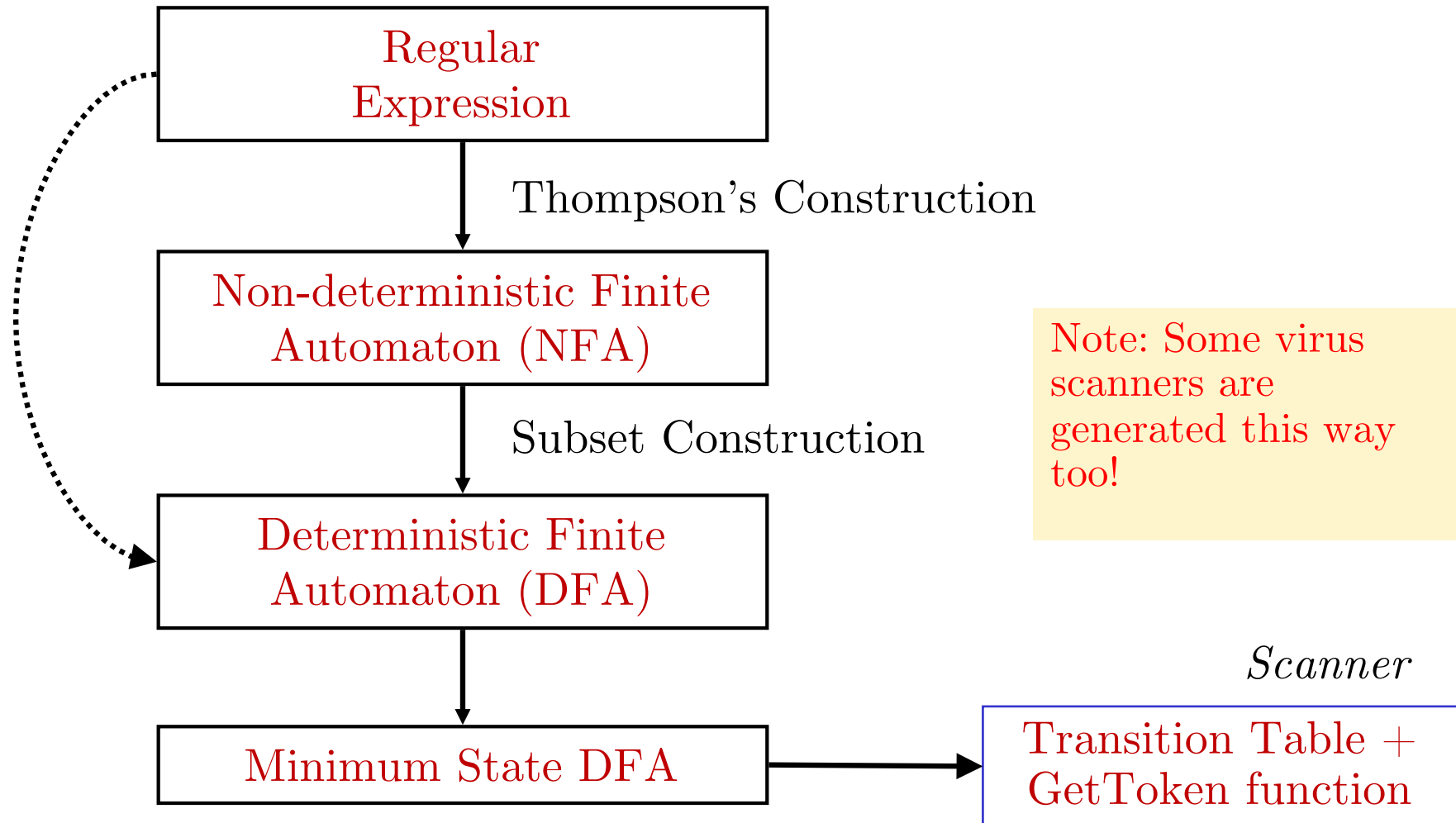
Implementation

- **Hand-crafted.** It's relatively easy to write a scanner by hand. However, the approach is ad-hoc and modification of hand-written scanners is more error-prone.
- **Lexical Analyser Generators** e.g. Flex. These are programs that accept a formal description of the lexical structure of a language (definitions of regular expressions) and proceed to generate the code for a lexical analyser (scanner) that returns an object (token) corresponding to the next matched regular expression.



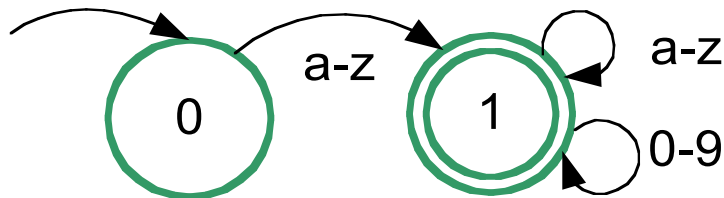
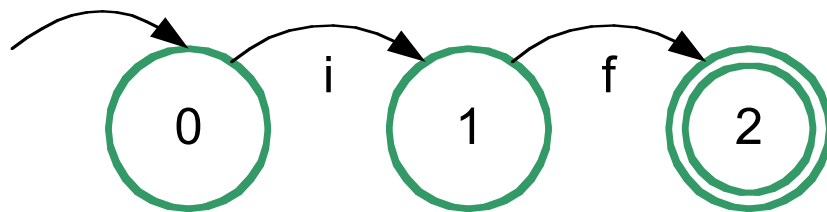
How does a lexical analyser generator perform a lexical analysis on its own input?

Lexical Analyser Generators



Finite Automata

Finite Automata (FA) are formal models for describing particular algorithms (or “machines”). We’ll use them to describe the process of recognising patterns in input strings to build scanners.



- **States** of the matching process (circles)
- **Transitions** between states (arrows)
- Matched input **Symbols** (labels on transitions)
- **Accepting states** of the matching process (double circles)
- **Start state** (has an unlabelled incoming arrow)

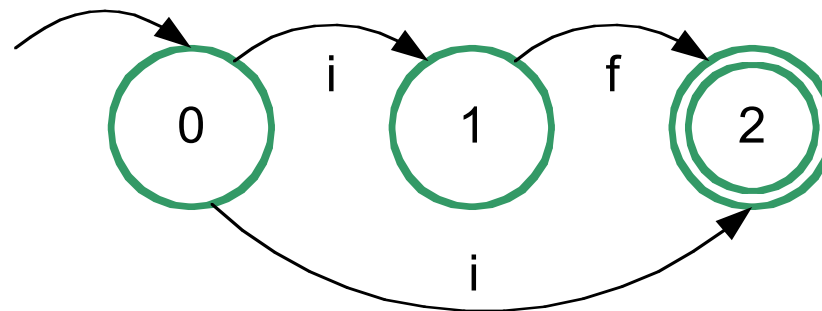
- Technically each non-accepting state should have a transition for every symbol. These missing transitions represent errors and are normally omitted from diagrams

DFA and NFA

- **Deterministic Finite Automata (DFA).** No two transitions leaving a state have the same symbol (fan-out of a symbol from a state is not allowed)



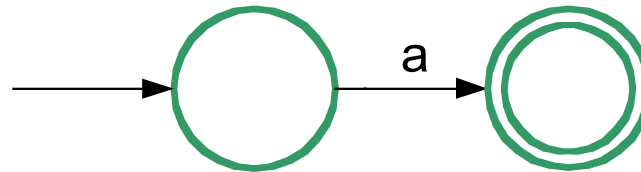
- **Non-deterministic Finite Automata (NFA).** Allow a choice of transitions out of a state (i.e. fan-out of a symbol from a state is allowed)



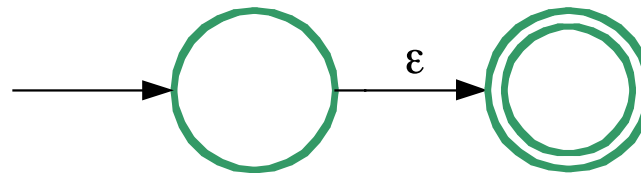
Regex to NFA

Thompson's Construction uses ϵ -transitions to “glue” together the automata for each part of a regular expression.

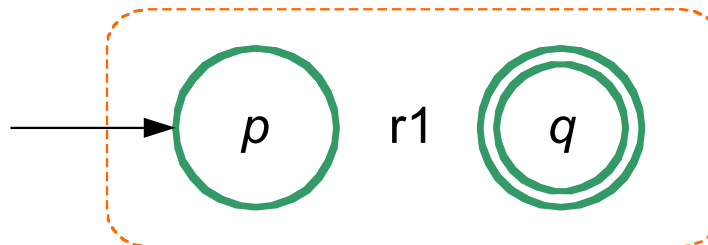
- **Symbol:** a of alphabet



- **Epsilon:** ϵ

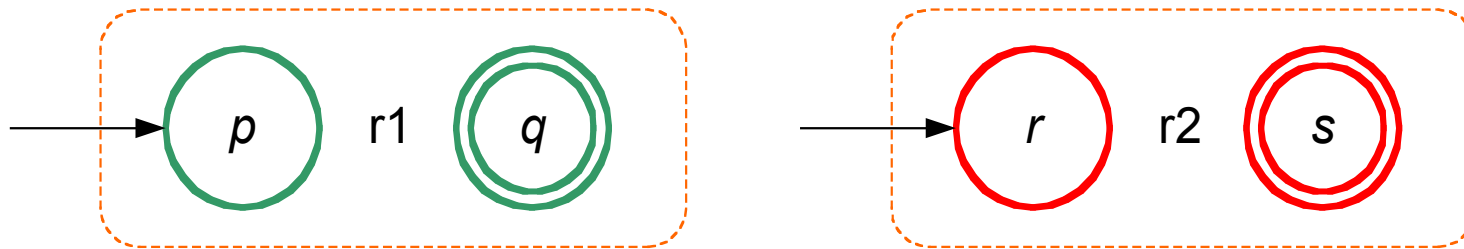


- **Grouping:** $(r1)$

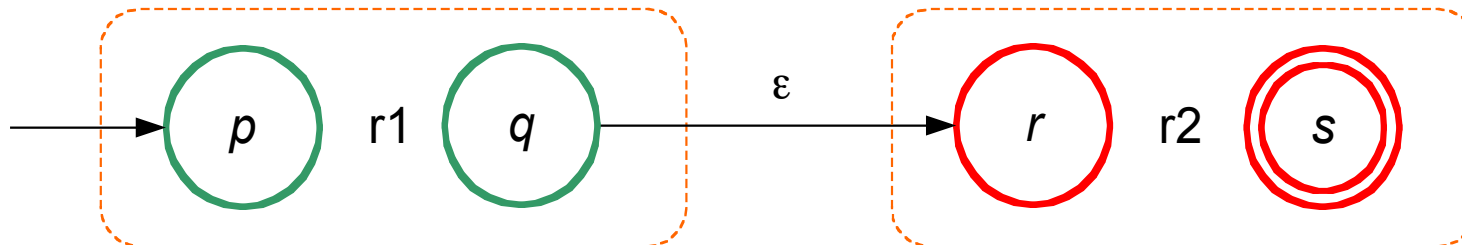


Concatenation: $r_1 r_2$

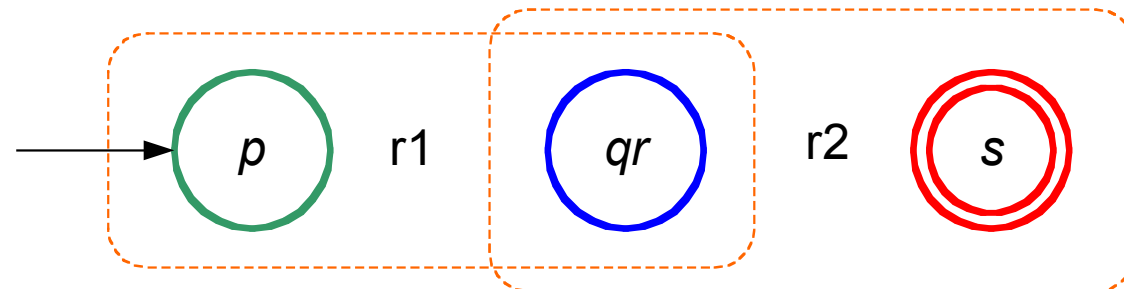
- Given



- Replace with

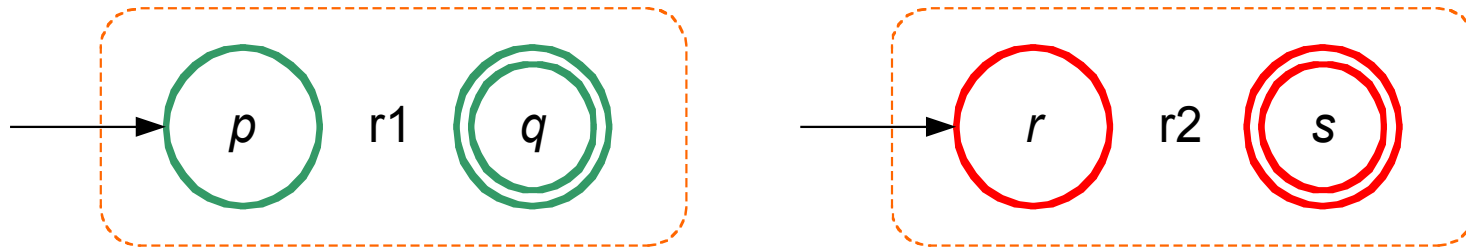


- or

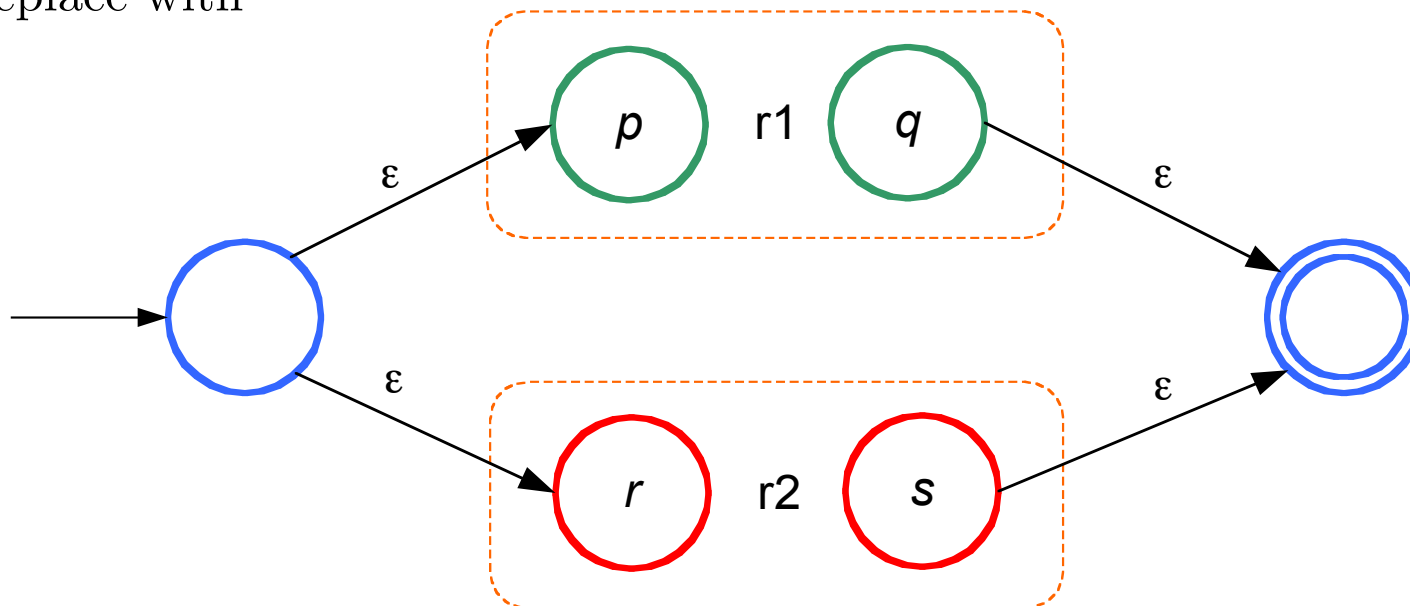


Alternation: $r1 \mid r2$

- Given

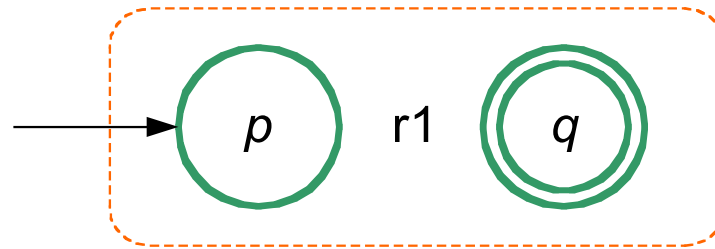


- Replace with

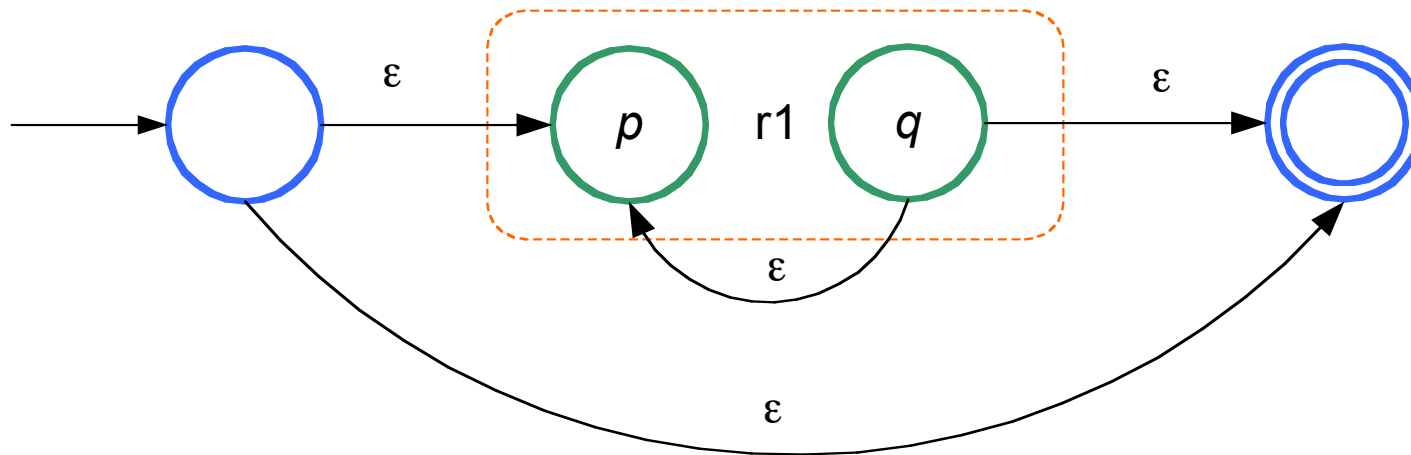


Repetition: r^*

- Given

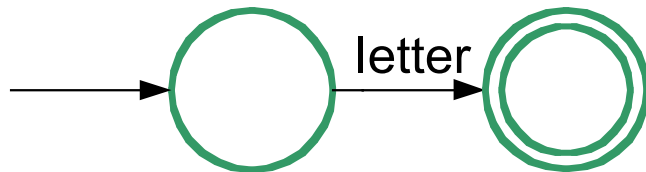


- Replace with

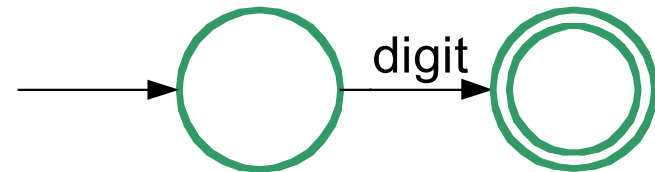


Example: Letter (Letter | Digit)*

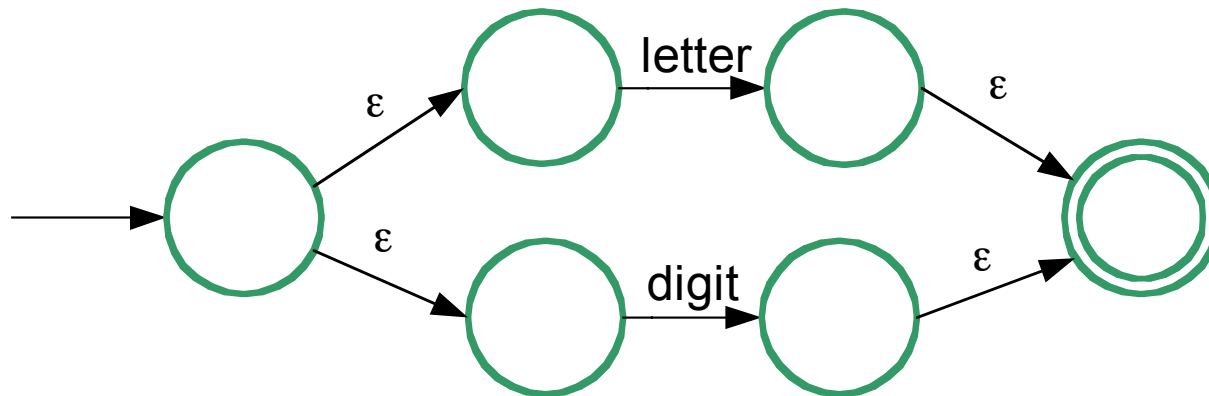
■ Letter



■ Digit

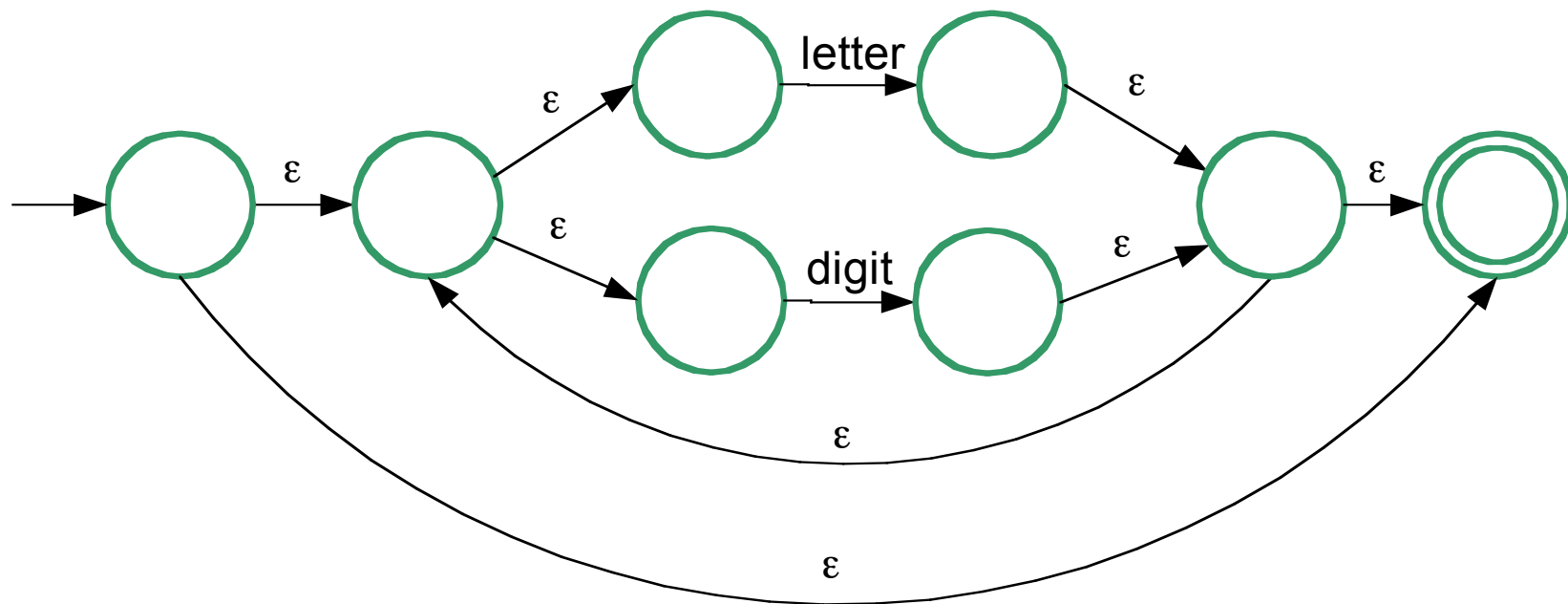


■ Letter | Digit



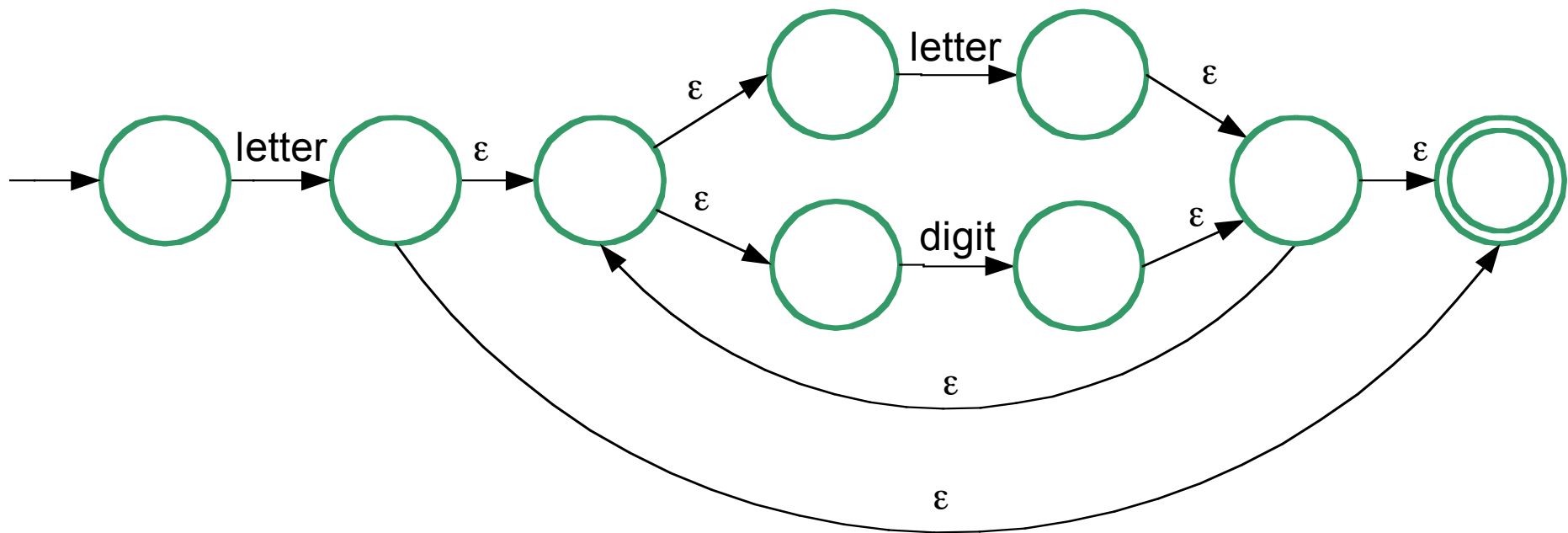
Example continued

- $(\text{Letter} \mid \text{Digit})^*$



Example completed

- Letter (Letter | Digit)*



NFA to DFA

- Although NFAs can be used as the basis of a scanner, their non-determinism needs to be simulated with an algorithm that can backtrack through every non-deterministic choice.
- For compilers, speed is important so we convert NFAs to DFAs using the Subset construction.
- Although DFAs are much faster than NFAs they require more memory (potentially 2^n states for an n -state NFA – see next slide). NFAs are sometimes used in editors that support Regex searches.
- **Subset Construction.** This eliminates ϵ -transitions and multiple outgoing transitions of the same character from states. States of the constructed DFA are “subsets” of the states of the NFA.

Complexity

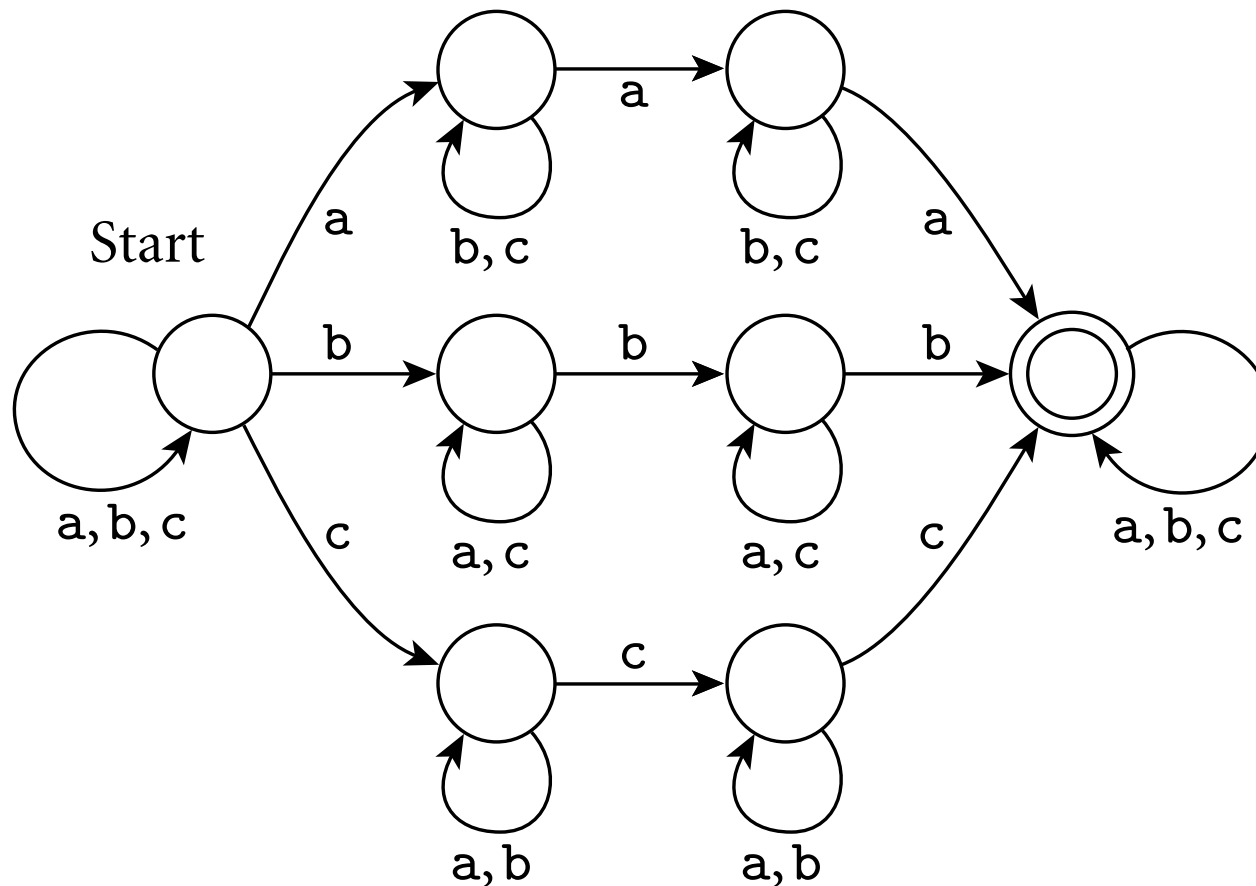
The table below summarises the worst case space & time requirements for an input string X using regular expression R .

	Space	Time
NFA	$O(\text{len } R)$	$O(\text{len } R * \text{len } X)$
DFA	$O(2^{\text{len } R})$	$O(\text{len } X)$

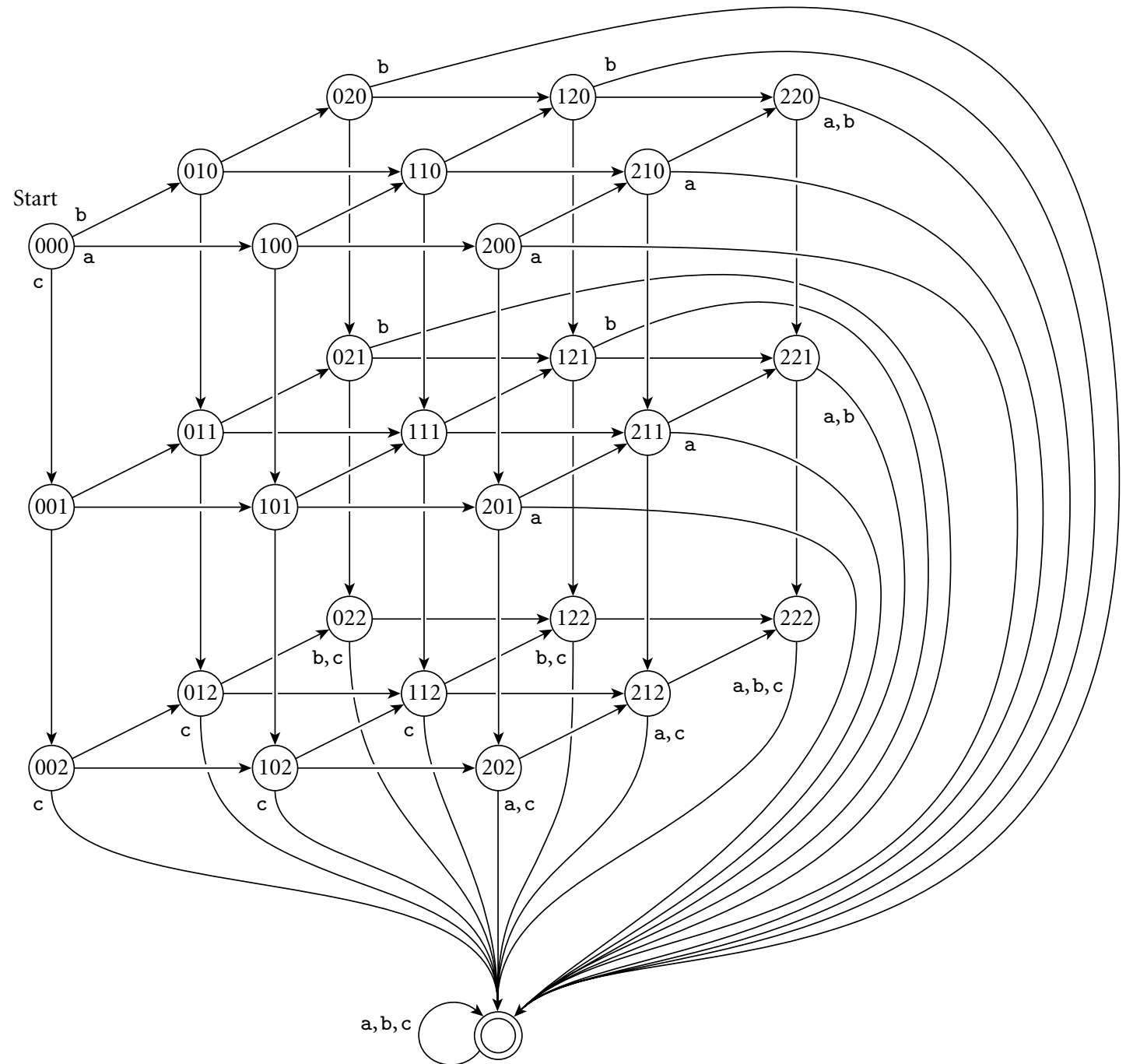
Fortunately for lexical analyser generators, the worst-case space requirements for generated DFAs rarely occur in practice and the speed performance of DFAs is great to have.

DFA State Explosion

What is the minimal DFA for the following NFA?



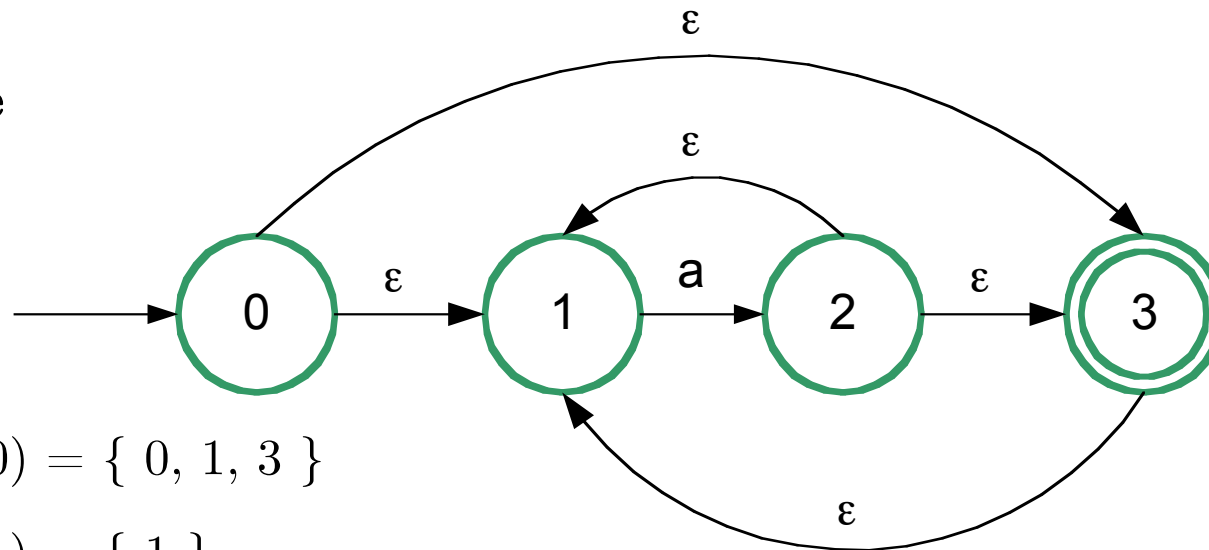
Minimal DFA is:



ϵ -Closures

- **ϵ -Closure** (s) = set of states reachable by zero or more ϵ -transitions from state s .
- **ϵ -Closure** ($\{s_1, \dots, s_N\}$) = union of ϵ -Closure (s_1), ..., ϵ -Closure (s_N)

- **Example**



ϵ -Closure (0) = $\{ 0, 1, 3 \}$

ϵ -Closure (1) = $\{ 1 \}$

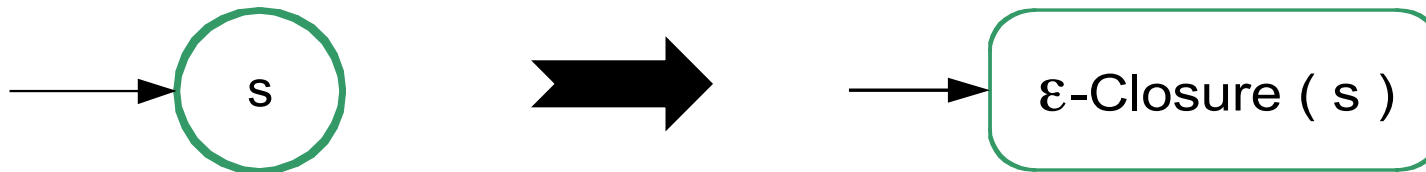
ϵ -Closure (2) = $\{ 1, 2, 3 \}$

ϵ -Closure (3) = $\{ 1, 3 \}$

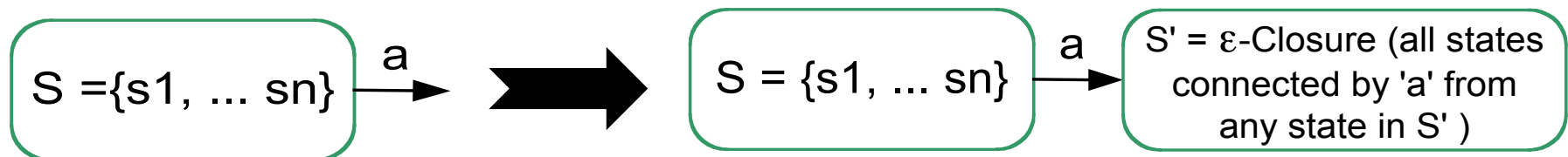
ϵ -Closure ($\{0, 2\}$) = $\{0, 1, 2, 3\}$

Subset Construction

- DFA start state = ϵ -Closure (NFA state state)



- foreach new subset state S of the DFA do
 foreach unique symbol 'a' leading out from any state of S do
 add a transition 'a' from S to S' where
 $S' = \epsilon\text{-Closure}$ (states reached by 'a' in 1 step)



- Mark subset states as accepting if any member state was accepting in the NFA

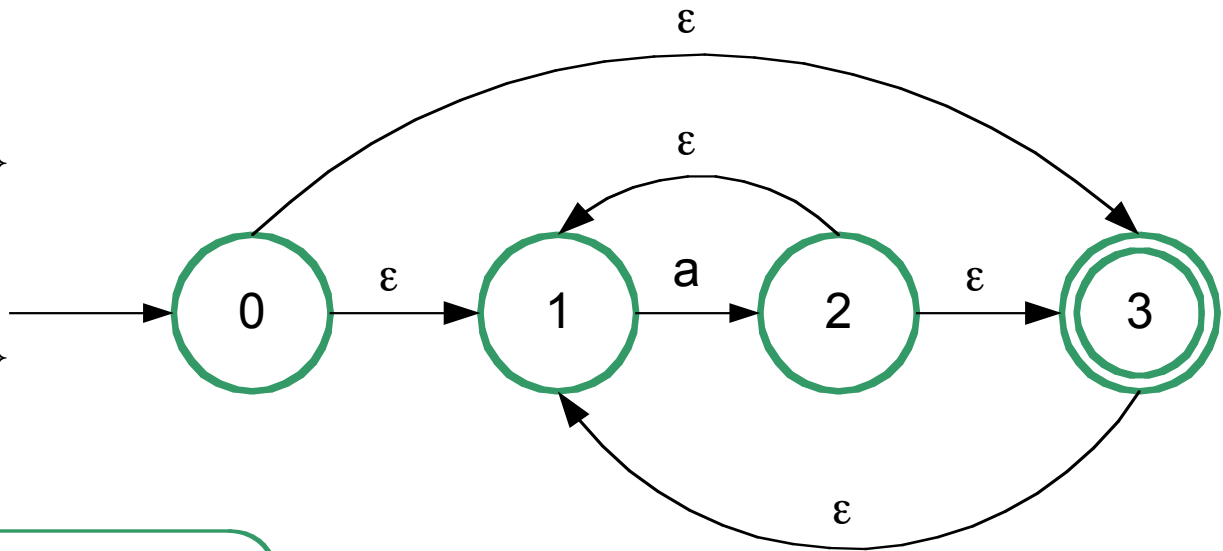
Example

ϵ -Closure (0) = { 0, 1, 3 }

ϵ -Closure (1) = { 1 }

ϵ -Closure (2) = { 1, 2, 3 }

ϵ -Closure (3) = { 1, 3 }



→ $\epsilon (0) = \{0, 1, 3\}$

→ $\{0, 1, 3\} \xrightarrow{a}$

→ $\{0, 1, 3\} \xrightarrow{a} \epsilon (2) = \{1, 2, 3\}$

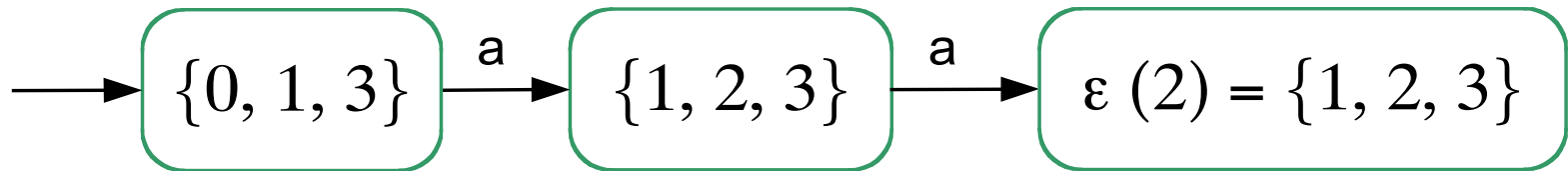
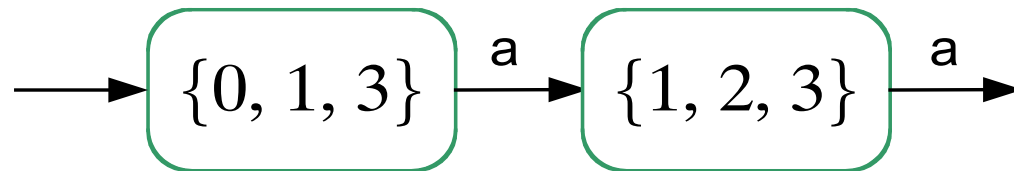
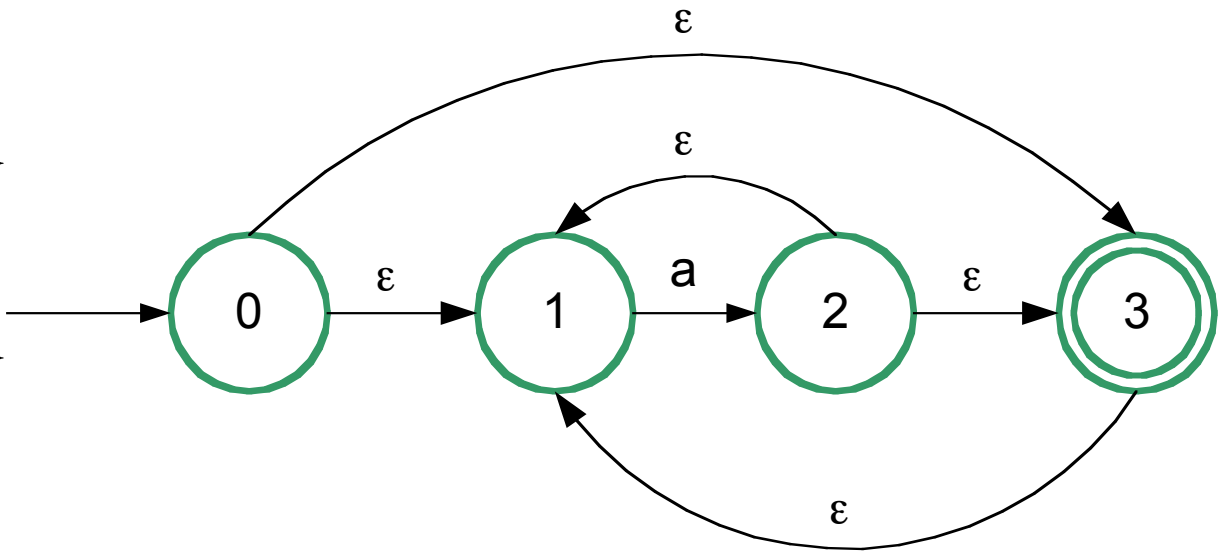
Example continued

ϵ -Closure (0) = { 0, 1, 3 }

ϵ -Closure (1) = { 1 }

ϵ -Closure (2) = { 1, 2, 3 }

ϵ -Closure (3) = { 1, 3 }



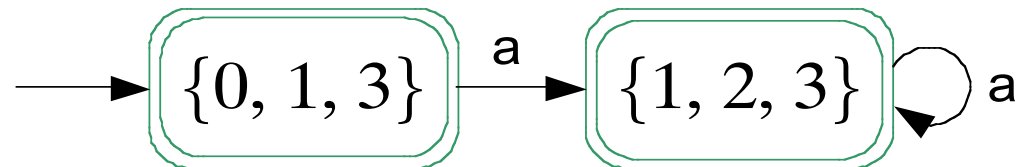
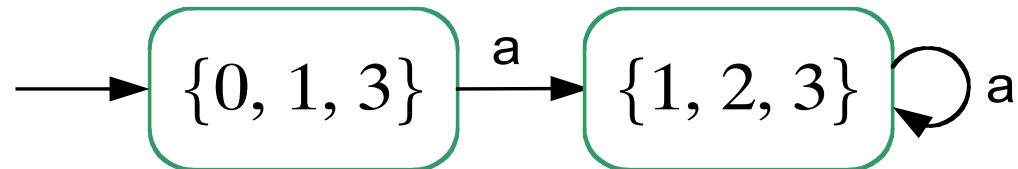
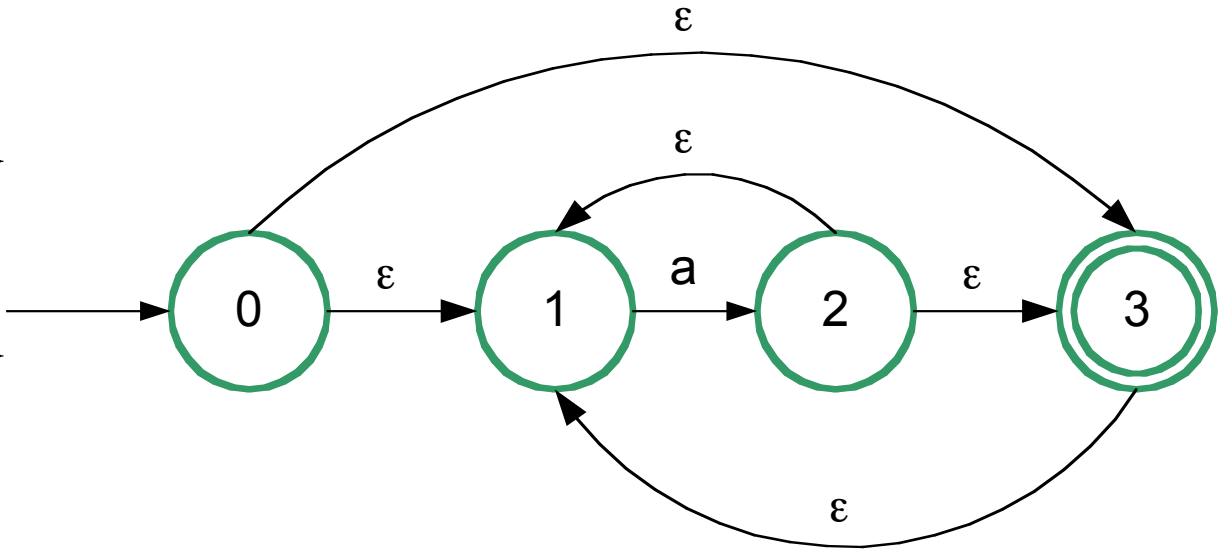
Example completed

ϵ -Closure (0) = { 0, 1, 3 }

ϵ -Closure (1) = { 1 }

ϵ -Closure (2) = { 1, 2, 3 }

ϵ -Closure (3) = { 1, 3 }



Minimal-State DFA (Optional Material)

Once a DFA is constructed from a NFA, we can optimise it by reducing it to its minimum size (i.e. minimal no. of states) with a further transformation.

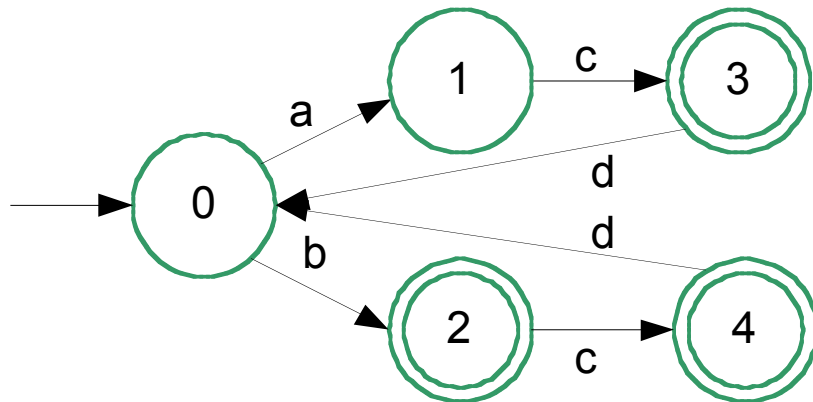
1. Partition DFA into (i) the set of accepting states A, (ii) set of non-accepting states N. Partitions = {A, N};
2. Let NewPartitions = { }
3. For each set S in Partitions do steps 4-5
4. For each state in S identify where each transition leads to i.e. leads either to the same set S or a different set.
5. Split S into new sets based on equivalence of their transitions.
 Add new sets to NewPartitions
6. Repeat steps 2-5 with Partitions=NewPartitions until no new split sets
7. Final non-splitable sets form new minimised states. A new minimised state is accepting if any member is accepting.
8. Preserve transitions based on original DFA

Example (Optional Material)

Partitioning:

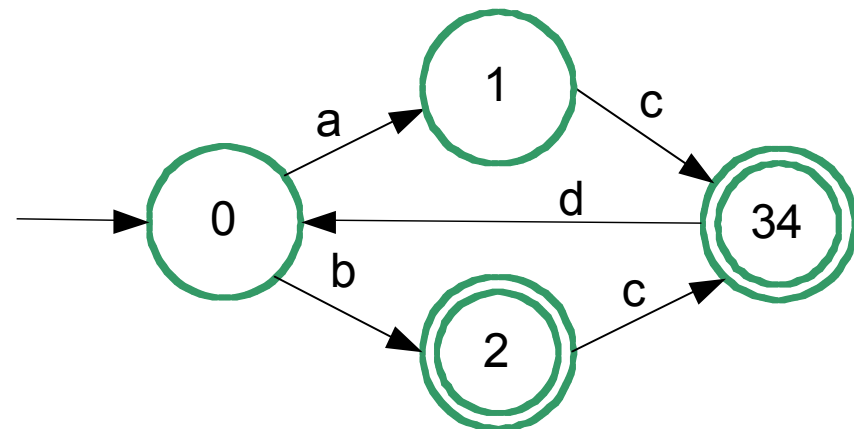
$A=\{2,3,4\}$ $N=\{0,1\}$

A	A	N	N	A	N
2	{c}		0	{b}	{a}
3		{d}	1	{c}	
4		{d}			



$A1=\{2\}$ $A2=\{3,4\}$ $N1=\{0\}$ $N2=\{1\}$

A2	A1	A2	N1	N2
3			{d}	
4			{d}	



Generated Lexical Analyser

Lexical analyser generators can encode DFAs as a 2D table:

DFA : Array [TotalStates][TotalChars] of State

and emit a simple `get_token` function for the parser to call, e.g.

```
def get_token():
    state = 0
    ch = getch()

    while not AcceptState[state] and not ErrorState[state]:
        state = DFA[state][ch]

    if AcceptState[state]:
        return Token[state] # return Token corresponding to state
    else return ErrorToken
```

Since transition tables are mostly empty, implementation techniques that compress tables are occasionally used by generators.

Miscellaneous

There are a number of questions about lexical analyser generators that we've conveniently omitted:

- *Are the DFAs for the Regex rules of a language composed with alternation?*
Yes
- *What happens on reaching an accepting state?* Typically we return a token for the appropriate regular expression.
- *What about token attributes?* A function to evaluate the attribute needs to be attached with the regular expressions for such tokens.
- *What happens on an error?* Typically return an Error token and/or backtrack.
- *How is the longest matching substring handled?* The get token function needs to make sure that we continue to make state transitions until we reach a state with no next state for the current symbol.
- *What about textual priority for regex rules?* The subset construction needs to ensure that the accepting state for the earliest rule is marked in our tables.

Summary

Lexical analysis plays a small but important role in compilers. It also illustrates the practical application of two interesting ideas in computing: regular expressions for pattern matching, and finite automata for representing algorithms for recognising such patterns.

These concepts have elegant and well established theories and for lexical analysis lead to very efficient scanners → good computing science!

For more details on lexical analysis and further exercises see:

[Cooper – Chapter 2]

[Appel – Chapter 2]

[Aho – Chapter 3]

Flex-Bison/C Example 1

scanner.l

%%

```
[0-9]+ { sscanf (yytext, "%d", &yyval) ; return INT; }
"+"    { return PLUS; }
"-"    { return MINUS; }
"*"    { return STAR; }
"="    { return EQ; }
"("    { return LPAR; }
")"    { return RPAR; }
```

- **yytext** holds matched chars, **yyleng** holds the number of chars matched.
- Return values (e.g. PLUS) correspond to tokens. Declared in bison grammar file.
- Set **yyval** to any value we want to associate with the matched token. **yyval** normally defined in a bison grammar file and is typically a **union**(defaults to **int**).
- **yytext** holds matched chars, **yyleng** holds the number of chars matched.
- **flex** generates a file called **lex.yy.c** containing the lexical analyser written in C. The lexical analyser will include a function called **yylex** which Bison's **yyparse** function will call to process the next token.

calc.y

```
%token INT PLUS MINUS STAR EQ LPAR RPAR
%left PLUS MINUS
%left STAR
```

Tokens given int values by default

Operators defined later have higher precedence than earlier operators

%%

```
prog:      expr EQ      { printf ("\n%d\n", $1); exit(0); } ;
```

```
expr:      expr PLUS   expr      { $$ = $1 + $3; }
          | expr MINUS expr      { $$ = $1 - $3; }
          | expr STAR   expr      { $$ = $1 * $3; }
          | INT          { $$ = $1; }
          | LPAR expr   RPAR      { $$ = $2; } ;
```

%%

```
#include "lex.yy.c"
```

```
int yerror (char *s) { }
```

```
int main () { return yyparse(); return 0; }
```

Import code generated from Flex definition file

Call generated parser

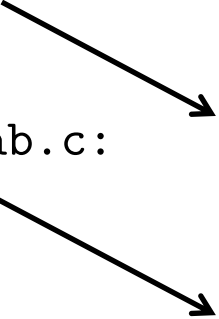
makefile

```
lex.yy.c:      scanner.l
               flex scanner.l

calc.tab.c:    lex.yy.c calc.y
               bison calc.y

calc:          calc.tab.c
               gcc calc.tab.c -ll -ly -w -o calc

clean:
               rm -f lex.yy.c calc.tab.c calc
```



ANTLR

- ANTLR is covered next week, but the definition of lexical tokens is similar to Flex e.g.

```
lexer grammar ElvisTokens;  
ID    : [a-zA-Z]+ ;      // Identifier  
INT   : [0-9]+ ;        // Integer  
EOL   : '\r'? '\n' ;     // End-Of-Line  
WS    : [ \t]+ -> skip ; // skip spaces and tabs  
MUL   : '*' ;  
DIV   : '/' ;  
ADD   : '+' ;  
SUB   : '-' ;
```

ANTLR

- Context-free Grammar

Import definitions of tokens

```
grammar Elvis;  
import ElvisTokens;  
  
prog : cmd+ ;  
  
cmd  : expr EOL  
      | ID '=' expr EOL  
      | EOL  
      ;  
  
expr : expr ('*' | '/') expr  
      | expr ('+' | '-') expr  
      | INT  
      | ID  
      | '(' expr ')'  
      ;
```