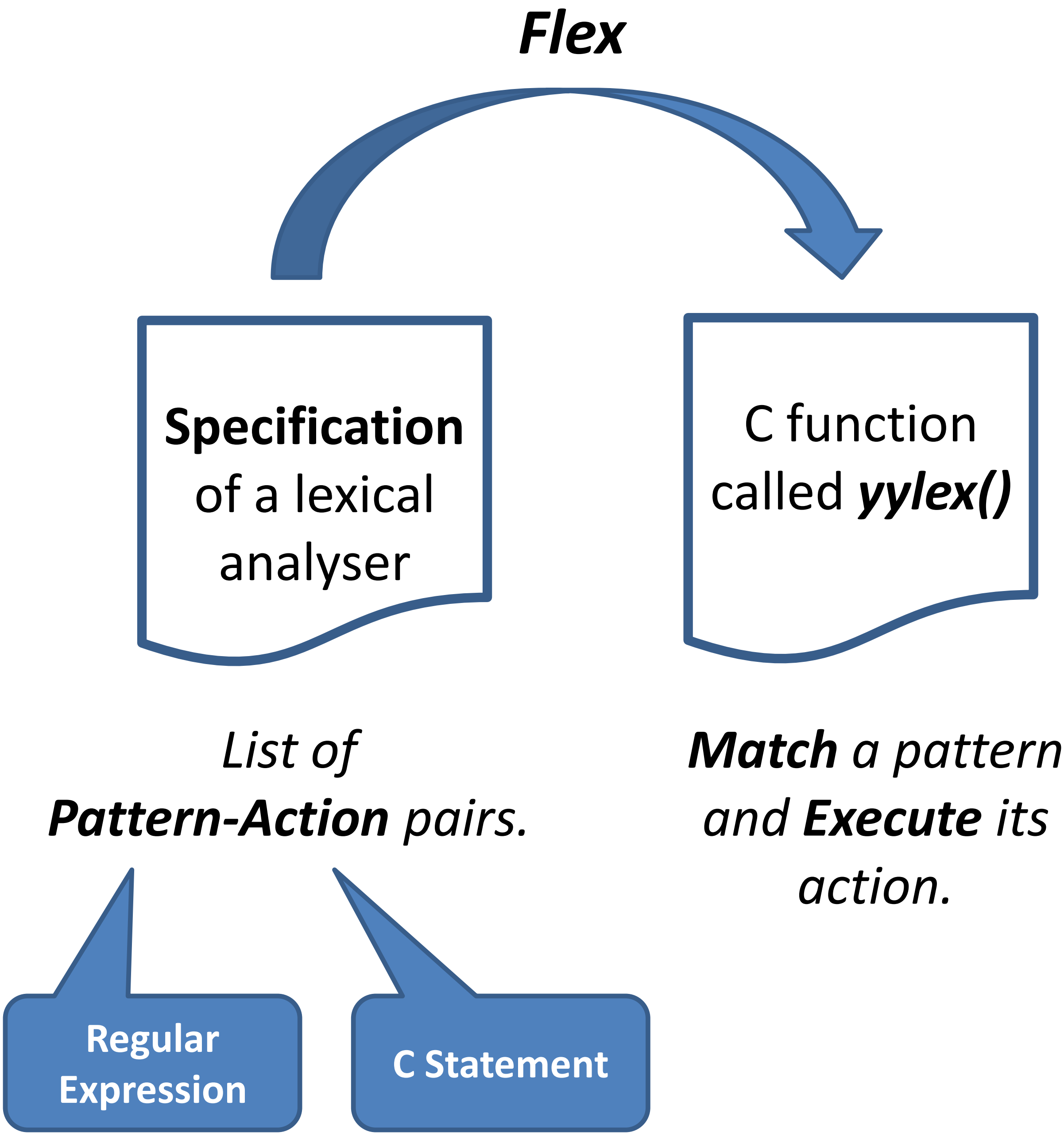
Lexical and Syntax Analysis

Flex, a Lexical Analyser Generator

Flex: a **fast lex**ical analyser generator



Input to ***Flex***

The structure of a ***Flex*** (*.lex*) file is as follows.

```
/* Declarations */

%%

/* Rules (pattern-action pairs) */

%%

/* C Code (including main function) */
```

Any text enclosed in /* and */ is treated as a **comment**.

What is a **rule**?

A **rule** is a pattern-action pair, written

pattern *action*

The **pattern** is (like) a regular expression. The **action** is a C statement, or a block of C statements in the form $\{\cdots\}$.

Example 1

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
/* No declarations */

%%

tom                 printf("jerry");
jerry               printf("tom");

%%

/* No main function */
```

Output of ***Flex***

Flex generates a C function

```
int yylex()
{
   ...
}
```

When *yylex()* is called:

1. a pattern that matches a **prefix** of the input text is **chosen**;

2. the matching text is **consumed**.

Output of ***Flex***

3. the **action** corresponding to the chosen pattern is **executed**;

4. if no pattern is chosen, a single character is consumed and echoed to output.

5. repeats until all input is consumed or an action executes a ***return*** statement.

Example 1, revisited

Replace all tom's with jerry's and vice-versa.

tomandjerry.lex

```
/* No declarations */

%%

tom             printf("jerry");
jerry           printf("tom");

%%

void main() {
    yylex();
}
```

Running Example 1

At a command prompt '>':

*> **flex** -o tomandjerry.c tomandjerry.lex*

*> **gcc** -o tomandjerry tomandjerry.c **-lfl***

> tomandjerry

jerry should be scared of tom.

tom should be scared of jerry.

Important!

Input

Output

Maximal munch!

Many patterns may match a prefix of the input. Which one does ***Flex*** choose?

- The one that matches the **longest** string.
- If different patterns match strings of the same length then the **first** pattern in the file is preferred.

What is a **pattern**? (Base cases)

Pattern	Meaning
x	Match the character ‘x’.
.	Match any character **except** a newline character (‘\n’).
[xyz]	Match either an ‘x’, ‘y’ or ‘z’.
[ad-f]	Match an ‘a’, ‘d’, ‘e’, or ‘f’.
[^A-Z]	Match any character **not** in the range ‘A’ to ‘Z’.
[a-z]{-}[aeiou]	Lower case consonants.
<<EOF>>	Matches end-of-file.

What is a **pattern**? (Inductive cases)

If p, p_1, p_2 are patterns then:

Pattern	Meaning
p_1p_2	Match a p_1 **followed by** a p_2.
$p_1\|p_2$	Match a p_1 **or** a p_2.
*p**	Match zero or more *p*'s.
p+	Match one or more *p*'s.
p?	Match zero or one *p*'s.
p{2,4}	At least 2 *p*'s and at most 4.
p{4}	Exactly 4 *p*'s.
(p)	Match a *p*, used to override precedence.
^p	Match a *p* at beginning of a line
p$	Match a *p* at end of a line

Pattern exercises

Characterise the strings matched by the following *Flex* patterns.

- *(a|b){5}*
- *[^ \n\r\t]+*
- *.|\n*
- *x.+y*

Escaping

Reserved symbols include:

. $ ^ [ ] - ? * + | ( ) / { } < >

Reserved symbols can be matched by enclosing them in double quotes or prefixing them with a backslash. For example:

Pattern	Meaning
"[xy]"	Match ‘[’ then ‘x’ then ‘y’ then ‘]’.
"+"*	Match zero or more ‘+’ symbols.
\"+	Match one or more " symbols.

Declarations

/* **Declarations** */

%%

/* **Rules** (pattern-action pairs) */

%%

/* **C Code** (including *main* function) */

What is a **declaration**?

A declaration may be:

- a **C declaration**, enclosed in *%{* and *%}*, visible to the action part of a rule.

- a **regular definition** of the form

name	*pattern*

introducing a new pattern *{name}* equivalent to *pattern.*

Example 2

```
%{
    int chars = 0;
    int lines = 0;
%}

%%

.           { chars++; }
\n          { lines++; chars++; }

%%

void main() {
    yylex();
    printf("%i %i\n", chars, lines);
}
```

Example 3

```
SPACE           [ \t\r\n]
WORD            [^ \t\r\n]+

%{
    int words = 0;
%}

%%

{SPACE}
{WORD}          { words++; }

%%

void main() {
    yylex();
    printf("%i\n", words);
}
```

yytext and ***yyleng***

The string matching a pattern is available to the action of a rule via the *yytext* variable, and its length via *yyleng*.

```
char* yytext;
int yyleng;
```

Global variables

Warning: the memory pointed to by *yytext* is destroyed upon completion of the action.

Example 4

inc.lex

```
DIGIT           [0-9]

%%

{DIGIT}+        {
                    int i = atoi(yytext);
                    printf("%i", i+1);
                }

%%

void main() {
    yylex();
}
```

Exercise 1

Give a Flex program that reverses each word occurring in the input.

Example input:

quick brown fox

Example output:

kciuq nworb xof

Tokenising using Flex

The idea is that *yylex()* returns **the next token**. This is achieved by using a ***return*** statement in the action part of a rule.

Some tokens have a **semantic value**, e.g. *NUM*, which by convention is returned via the global variable *yylval*.

```
int yylval;
```

Global variable

Example 5

nums.lex

```
%{
   typedef enum { END, NUM }  Token;
%}

%%

[^0-9]            /* Ignore */
[0-9]+            {
                     yylval = atoi(yytext);
                     return NUM;
                  }
<<EOF>>           { return END; }

%%

void main() {
   while (yylex() != END)
      printf("NUM(%i)\n", yylval);
}
```

The type of ***yylval***

By default *yylval* is of type ***int***, but it can be overridden by the user. For example:

```
union {
    int number;
    char* string;
} yylval;
```

Now *yylval* can either hold a number or a string.

NOTE: When interfacing ***Flex*** and ***Bison***, the type of *yylval* is defined in the ***Bison*** file using the *%union* option.

Start conditions and states

If *p* is a pattern then so is *<s>p* where *s* is a state. Such a pattern is only **active** when the scanner is in state *s*.

Initially, the scanner is in state *INITIAL*. The scanner **moves** to a state *s* upon execution of a *BEGIN(s)* statement.

Inclusive states

An **inclusive state** *S* can be declared as follows.

```
%s S
```

When the scanner is in state *S* any rule with start condition *S* **or** no start condition is active.

Exclusive states

An **exclusive state** *S* can be declared as follows.

```
%x S
```

When the scanner is in state *S* **only** rules with the start condition *S* are active.

Example 6

strip.lex

```
%x   COM          /* In comment */

%%

"/*"              { BEGIN(COM); }
<COM>"*/"         { BEGIN(INITIAL); }
<COM>.|\n         /* Ignore */

%%

void main() {
    yylex();
}
```

Exercise 2

Consider the following payroll.

Wayne Rooney,Footballer,13000000
David Cameron,Prime Minister,142500
Joe Bloggs,Programmer,40000

Write a ***Flex*** specification that takes a payroll and outputs the sum of the salaries.

Variants of ***Flex***

There are ***Flex*** variants available for many languages:

Language	**Tool**
C++	Flex++
Java	JLex
Haskell	Alex
Python	PLY
Pascal	TP Lex
*	ANTLR

Summary

- ***Flex*** converts a list of **pattern-action** pairs into C function called *yylex()*.
- Patterns are similar to **regular expressions**.
- The idea is that *yylex()* identifies and returns the **next token** in the input.
- Gives a declarative (**high level**) way to define lexical analysers.

THE THEORY BEHIND FLEX

"Under the hood"

Outline

Automatic conversion of regular expressions to efficient string-matching functions:

- **Step 1:** RE $\longrightarrow$ NFA
- **Step 2:** NFA $\longrightarrow$ DFA
- **Step 3:** DFA $\longrightarrow$ C Function

Acronym	Meaning
RE	Regular Expression
NFA	Non-deterministic Finite Automaton
DFA	Deterministic Finite Automaton

STEP 1: RE $\longrightarrow$ NFA

Thompson’s construction

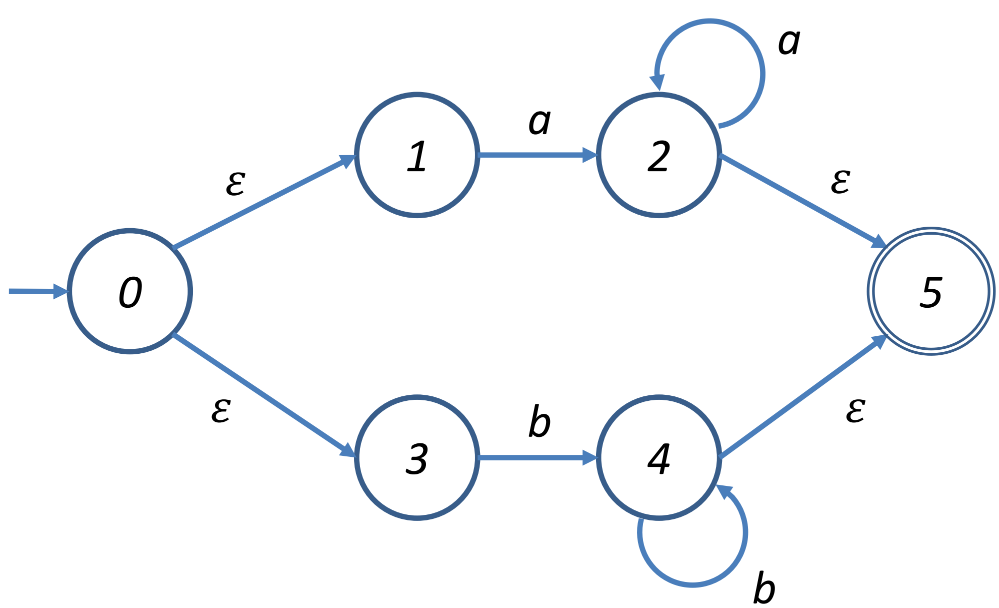
Thompson’s construction

An algorithm for tuning **any** regular expression into an NFA.

Example **input**:

$$a \cdot a^{*} \mid b \cdot b^{*}$$

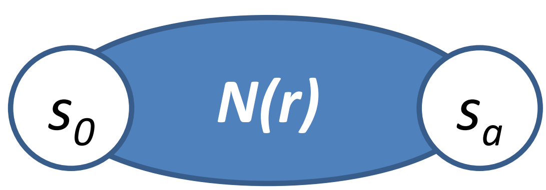
Example **output**:



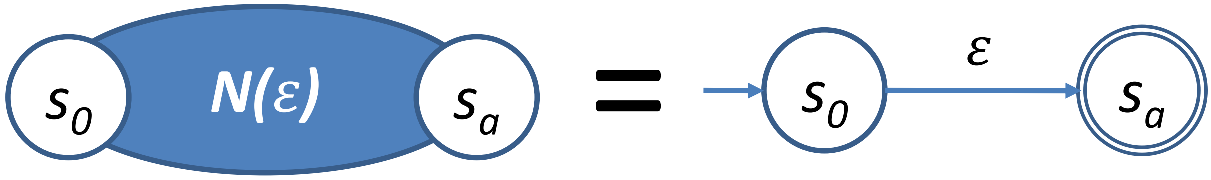
Thompson's construction: **Notation**

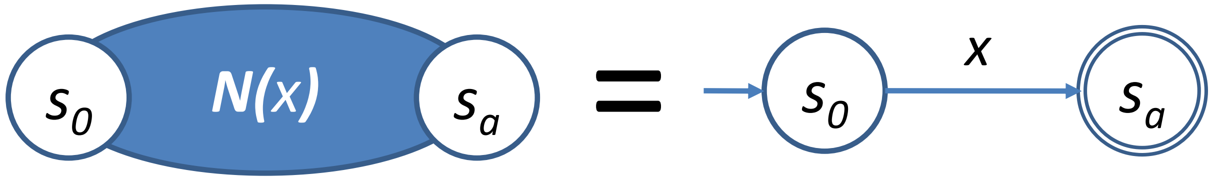
Let *N(r)* be the NFA accepting exactly the set of strings in *L(r)*.

We abstractly represent an NFA *N(r)* with start state s_0 and final state s_a by the diagram:



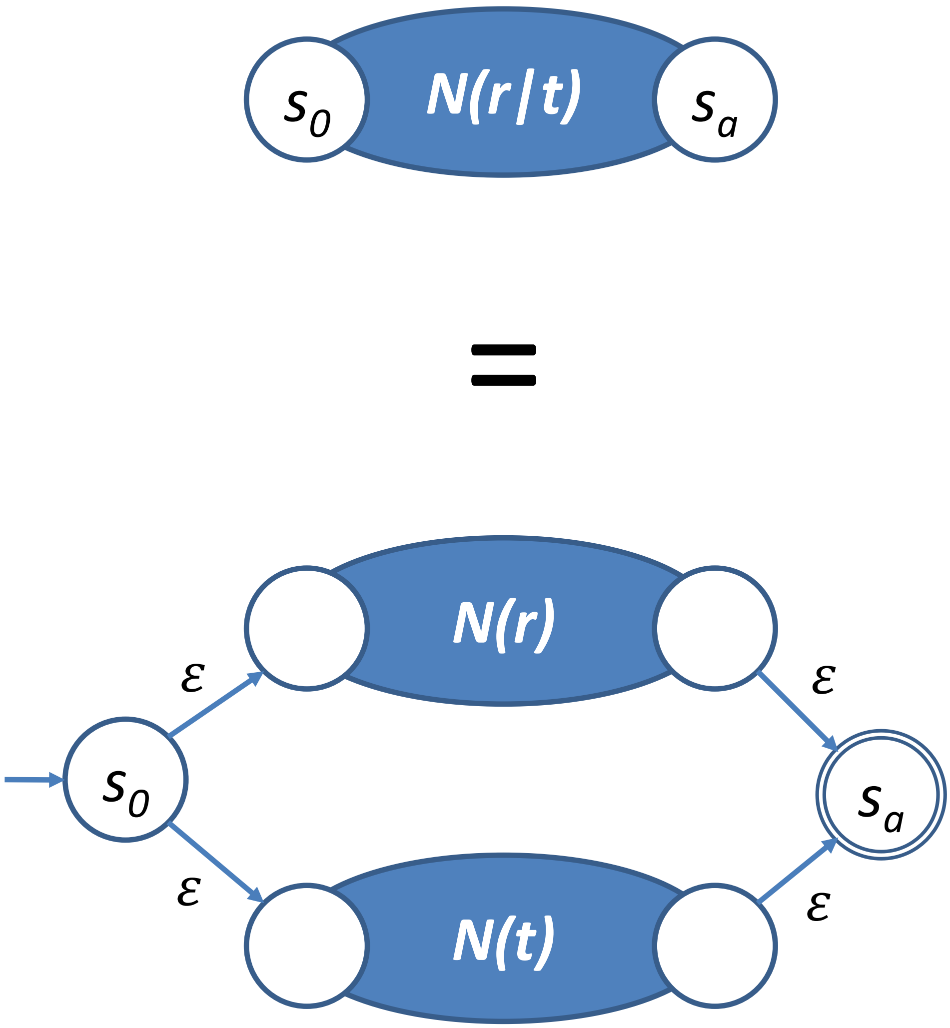
Thompson's construction: **Base cases**



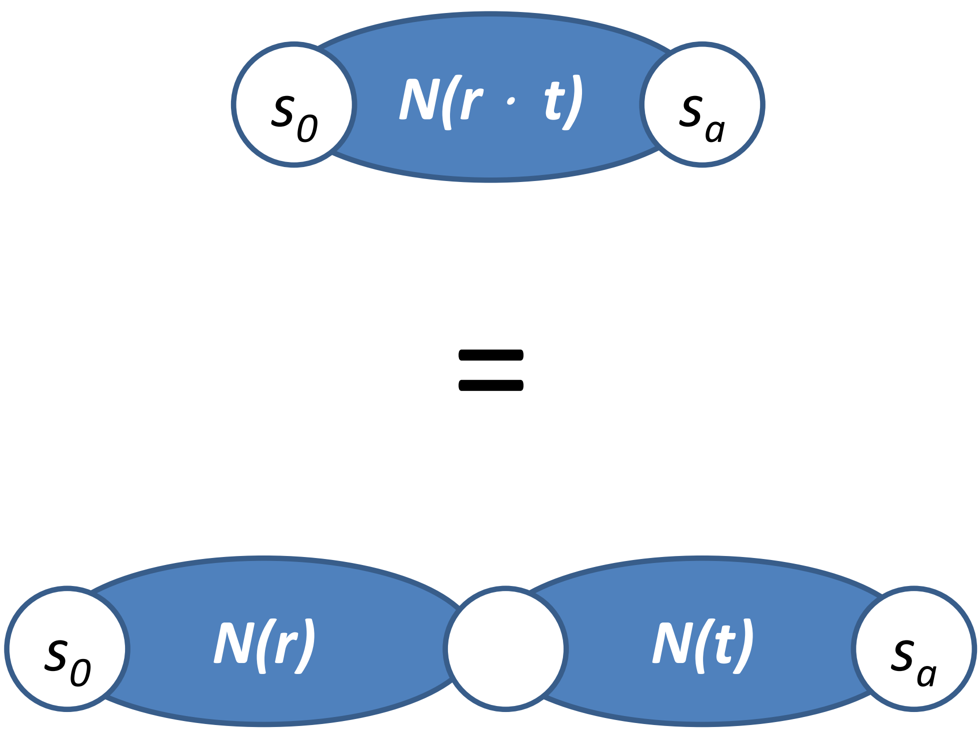


where $x \in \sum$

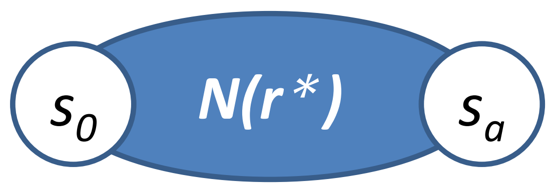
Thompson's construction: **Choice**



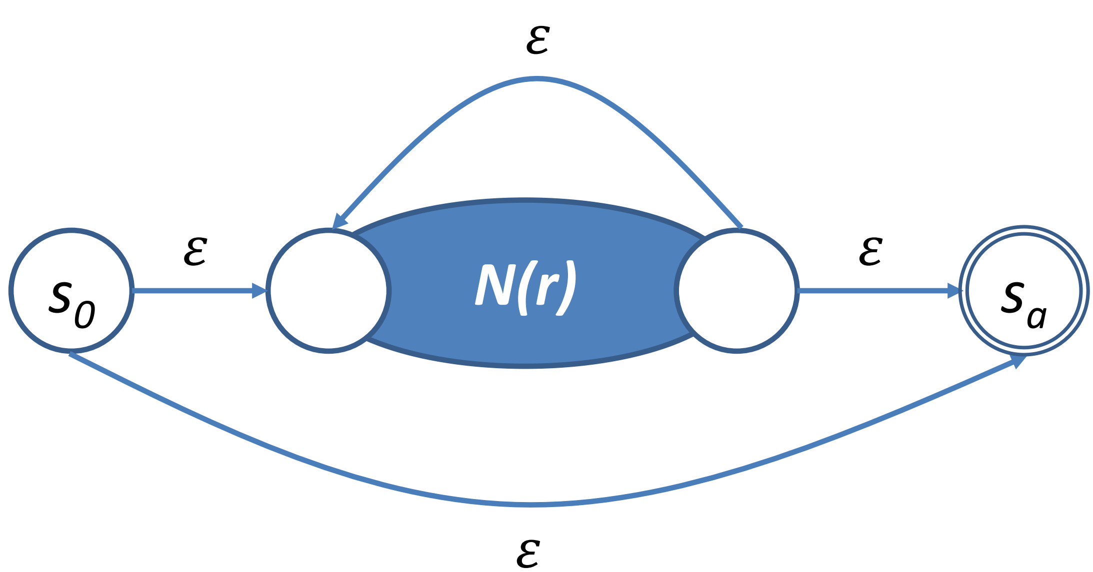
Thompson's construction: **Sequence**



Thompson’s construction:
Kleene closure



=



Exercise 3

Apply Thompson's construction to the following regular expression.

$$((a \cdot b) | c)^*$$

Problem with NFAs

It is not straightforward to turn an NFA into an efficient matcher because:

- There may be many possible next-states for a given input.
- Which one do we choose?
- Try them all?

Idea: convert an NFA into a DFA: a DFA can be easily converted into an efficient executable program.

STEP 2: NFA ⟶ DFA

The subset construction.

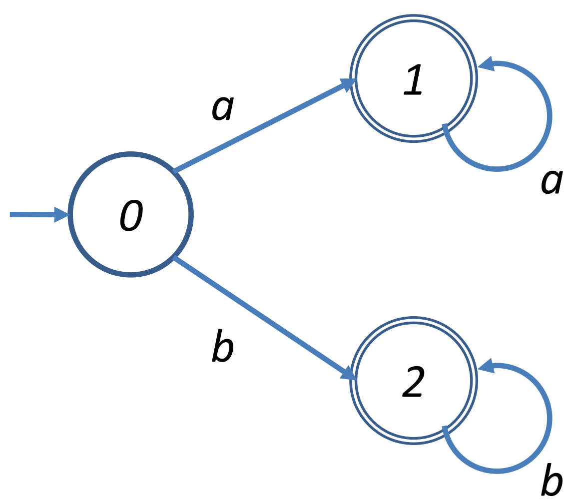
What is a DFA?

A deterministic finite automaton (DFA) is an NFA in which

- there are no ε transitions, and
- for each state *s* and input symbol *a* there is **at most one** transition out of *s* labelled *a*.

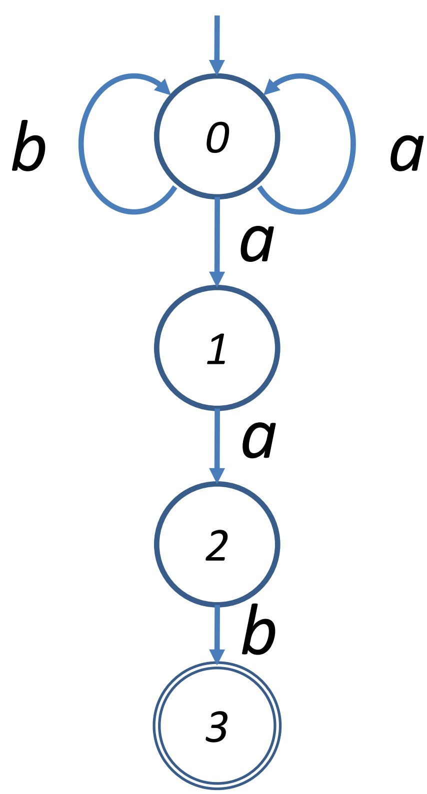
Example of a DFA

The following DFA accepts exactly the strings that match the regular expression $a \cdot a^* \mid b \cdot b^*$.



NFA → DFA: key observation

After consuming an input string, an NFA can be in be in one of a **set of states**. Example 3:



Input	States
aa	0, 1, 2
aba	
aab	
aaba	
ε	

NFA → DFA: key idea

Idea: construct a DFA in which each state corresponds to a **set** of NFA states.

After consuming $a_1 \cdots a_n$ the DFA is in a state which corresponds to the set of states that the NFA can reach on input $a_1 \cdots a_n$.

Example 3, revisited

Create a DFA state corresponding to each **set** of NFA states.

Input	NFA States	DFA State
aa	0, 1, 2	A
aba	0,1	B
aab	0,3	C
aaba	0,1	B
ε	0	D

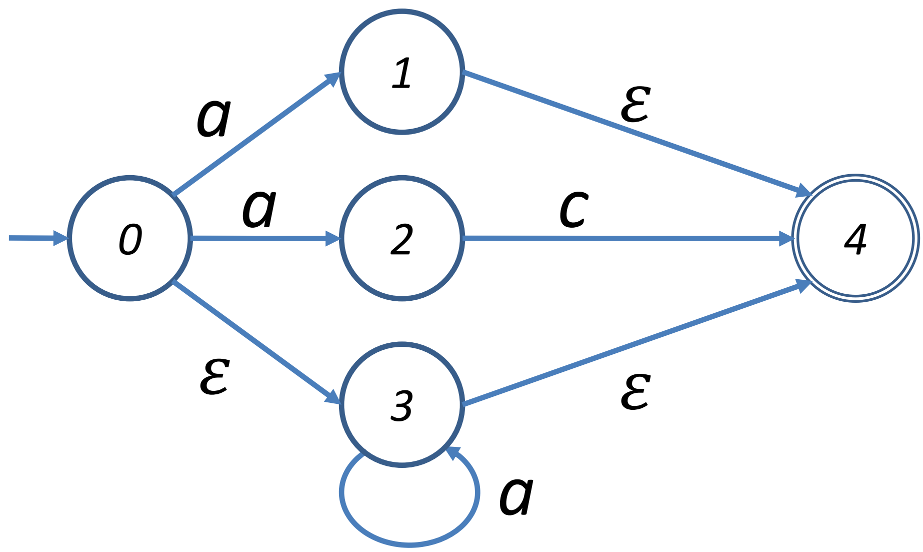
Question: which states would be initial and final DFA states?

Notation

Operation	Description
ε-closure(s)	Set of NFA states reachable from NFA state *s* on zero or more *ε*-transitions.
ε-closure(T)	$\bigcup_{s \in T}$ *ε-closure(s)*
move(T, a)	Set of NFA states to which there is a transition on symbol *a* from some state *s* in *T*.

Exercise 4

Consider the following NFA.



Compute:

- *ε-closure(0)*
- *ε-closure({1, 2})*
- *move({0,3}, a)*
- *ε-closure(move({0,3}, a))*

Subset construction: **input** and **output**

Input: an NFA *N.*

Output: a DFA *D* accepting the same language as *N.* Specifically, the **set of states** of *D*, termed D_{states}, and its **transition function** D_{tran} that maps any state-symbol pair to a next state.

Subset construction: **input** and **output**

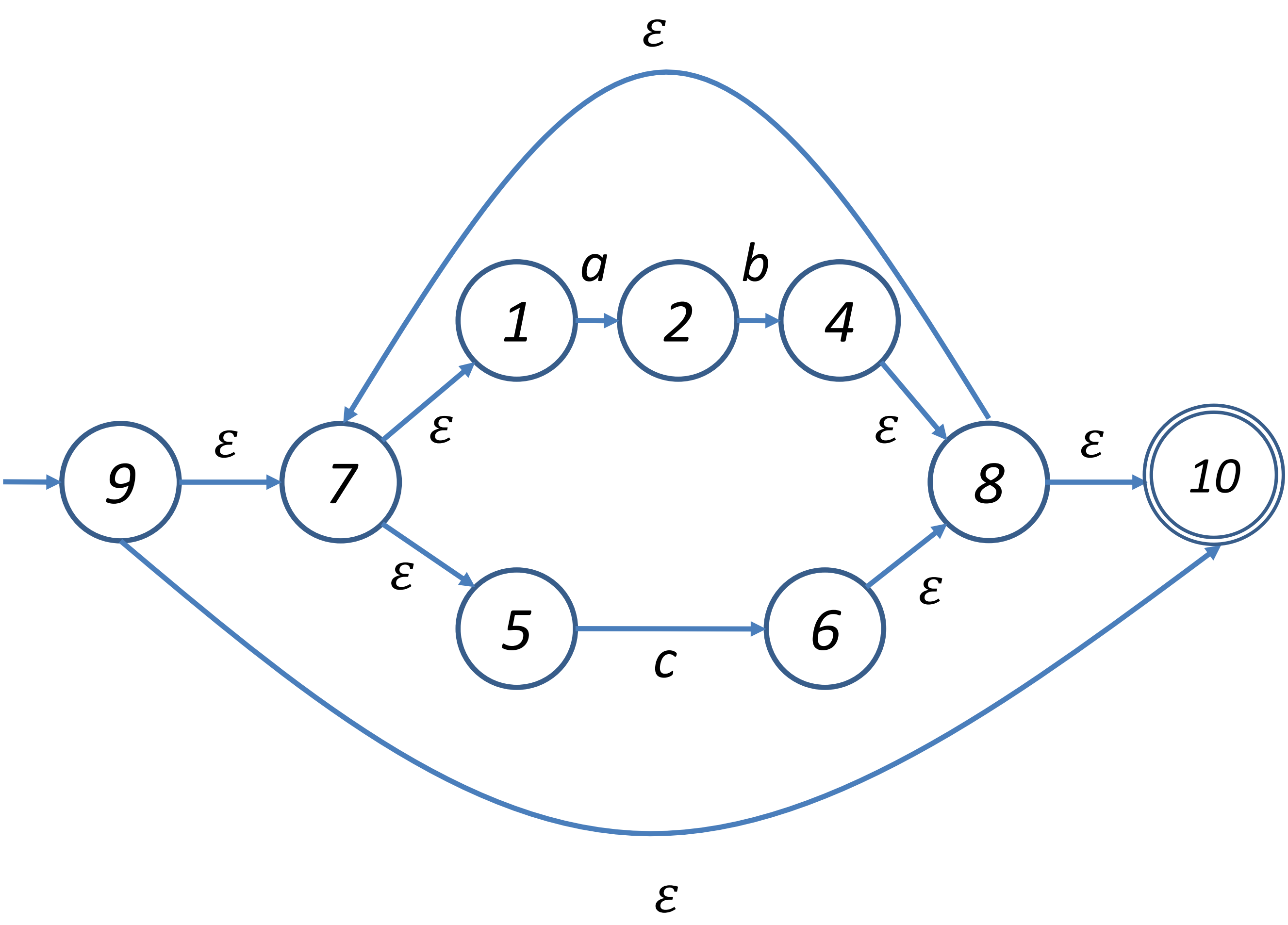
- Each state in *D* is denoted by a **subset** of *N*'s states.
- To ensure termination, every state is either **marked** or **unmarked**.
- **Initially**, D_{states} contains a single unmarked start state $\varepsilon\text{-}closure(s_0)$ where s_0 is the start state of *N*.
- The **accepting states** of *D* are the states that contain at least one accepting state of *N*.

Subset construction: **algorithm**

while (there is an unmarked
state T in D_{states}) {
  mark T;
  for (each input symbol a) {
    $U = \varepsilon\text{-}closure(move(T, a))$;
    $D_{tran}[T, a] = U$
    if (U is not in D_{states})
      add U as unmarked state to D_{states};
  }
}

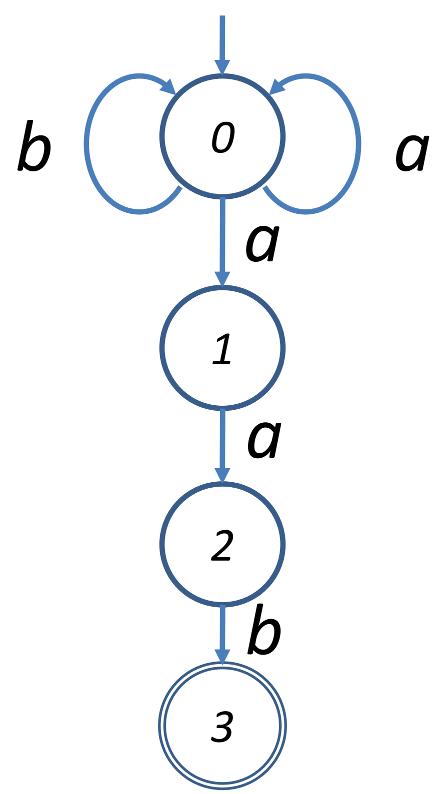
Exercise 5

Convert the following NFA into a DFA by applying the subset construction algorithm.



Homework Exercise

Convert the following NFA into a DFA by applying the subset construction algorithm.



Exercise 6

It is **not obvious** how to simulate an NFA in **linear time** with respect to the length of the input string.

But it may be converted to a DFA that can be simulated **easily** in linear time.

What's the catch? Can you think of any problems with the DFA produced by subset construction?

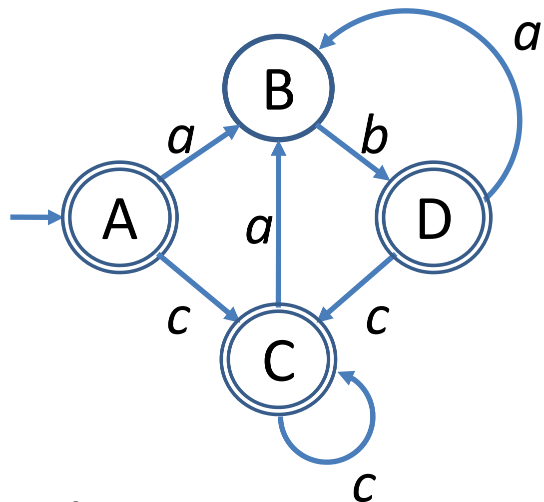
Caveats

- Number of DFA states **could be exponential** in number of NFA states!
- DFA produced is **not minimal** in number of states. (Can apply a minimisation algorithm.)

STEP 3: DFA ⟶ C CODE

Exercise 7

Implement the DFA



as a C function

```
int match(char *next) {
   ...
}
```

returning *1* if the string pointed to by *next* is accepted by the DFA and *0* otherwise.

```
int match(char* next) {
  goto A;

  A:    if (*next == '\0') return 1;
        if (*next == 'a')  { next++; goto B; }
        if (*next == 'c')  { next++; goto C; }
        return 0;

  B:    if (*next == '\0') return 0;
        if (*next == 'b')  { next++; goto D; }
        return 0;

  C:    if (*next == '\0') return 1;
        if (*next == 'a')  { next++; goto B; }
        if (*next == 'c')  { next++; goto C; }
        return 0;

  D:    if (*next == '\0') return 1;
        if (*next == 'a')  { next++; goto B; }
        if (*next == 'c')  { next++; goto C; }
        return 0;
}
```

SUMMARY

Summary

Automatically converting regular expressions into efficient C code involves three main steps:

1. **RE → NFA**
   (Thompson's Construction)

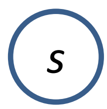
2. **NFA → DFA**
   (e.g. Subset Construction)

3. **DFA → C Function**
   (Straightforward)

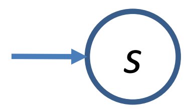
APPENDIX

What is an NFA?

A directed graph with **nodes** denoting **states**

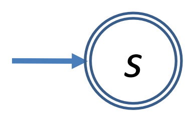


A **state** *s*



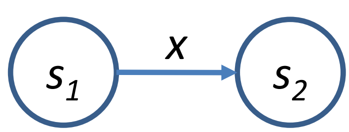
The **start state** *s*

An **accepting state** *s*



The **start state** *s* that is also an **accepting** state

and **edges** labelled with a symbol $x \in \sum \cup \{\varepsilon\}$ denoting **transitions**



Meaning of an NFA

A string $x_1x_2...x_n$ is **accepted** by an NFA if there is a path labelled $x_1,x_2,...,x_n$ (including any number of ε transitions) from the **start** state to an **accepting** state.