Lexical analysis-II

HW1 discussion

REs to NFA

NFA to DFA

lex intro

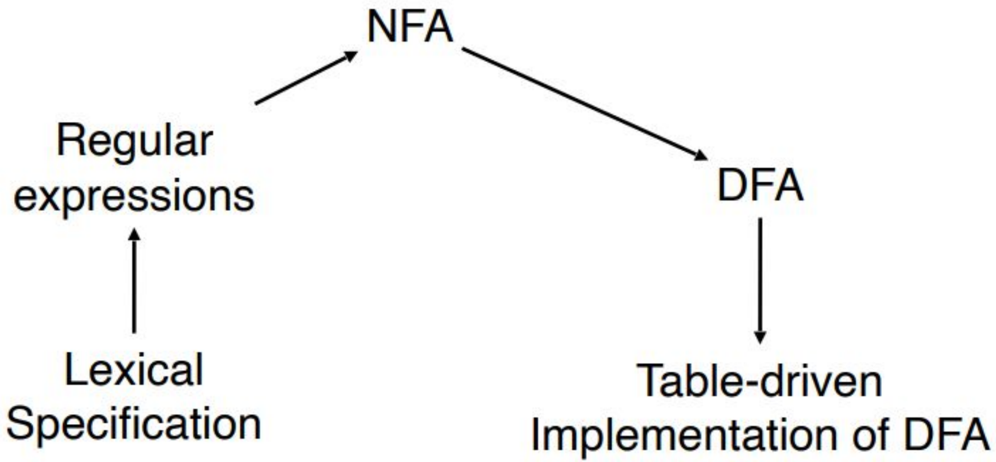
HW-2 and Project-1 info

Content is copied from:

- https://web.stanford.edu/class/cs143/lectures/lecture04.pdf
- Engineering a Compiler by Cooper and Torczon, 2nd Ed. ch. 1 and sec. 2.1-2.4
- https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

Convert Regular Expressions to Finite Automata

High level sketch



https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Regular Expressions in Lexical Specification

a specification for the predicate

$$s \in L(R)$$

- But a yes/no answer is not enough!
- Instead: partition the input into tokens
- We will adapt regular expressions to this goal

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Notation

There is variation in regular expression notation

- Union: A + B ≡ A | B
- Option: A + ε ≡ A?
- Range: ‘a’+’b’+…+’z’ ≡ [a-z]
- Excluded range: complement of [a-z] ≡ [^a-z]

Notation	Definition
.	**any character**
*****	**0 or more times**
+	**1 or more times**
?	**0 or 1 time**
{n}	**Exactly n number of times**
{n,}	**At least n times**
{n,m}	**At least n but not more than m times**
`[ ]`	`any single char inside brackets`
`[^ ]`	`none of the single chars inside brackets`
$	**when used as last char, match the end of the line**

Lexical Specification → Regex in five steps

1. Write a regex for each token

- `Number = digit +`
- `Keyword = ‘if’ + ‘else’ + …`
- `Identifier = letter (letter + digit)*`
- `OpenPar = ‘(‘`
- …

2. Construct R, matching all lexemes for all tokens

```
R     = Keyword + Identifier + Number + …
      = R1 + R2 + …
```

`(This step is done automatically by tools like flex)`

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

```
3. Let input be x1…xn
for 1 ≤ i ≤ n check
     x1…xi ∈ L(R)

4. If success, then we know that
     x1…xi ∈ L(Rj) for some j

5. Remove x1…xi from input and go to (3)
```

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Ambiguity

There are ambiguities in the algorithm

- How much input is used? What if

  x1…xi ∈ L(R) and also

  x1…xK ∈ L(R)

- Rule: Pick longest possible string in L(R)
  - Pick k if k > i
  - The “maximal munch”

Which token is used? What if

x1…xi ∈ L(Rj)

and

x1…xi ∈ L(Rk)

- Rule: use rule listed first
  - Pick j if j < k
  - E.g., treat “if” as a keyword, not an identifier

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Error handling

What if No rule matches a prefix of input?

- `Problem: Can't just get stuck …`

- `Solution:`

– `Write a rule matching all "bad" strings`

– `Put it last (lowest priority)`

Regular expressions provide a concise notation for string patterns

• Use in lexical analysis requires small extensions

– To resolve ambiguities

– To handle errors

• Good algorithms known

– Require only single pass over the input

– Few operations per character (table lookup)

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Finite automata again

Regular expressions = specification

Finite automata = implementation

For an input s

- FA accepts s if there is any computational path that accepts a string.

Deterministic Finite Automata (DFA)

– Exactly one transition per input per state

– No ε-moves

Nondeterministic Finite Automata (NFA)

– Can have zero, one, or multiple transitions for one input in a given state

– Can have ε-moves

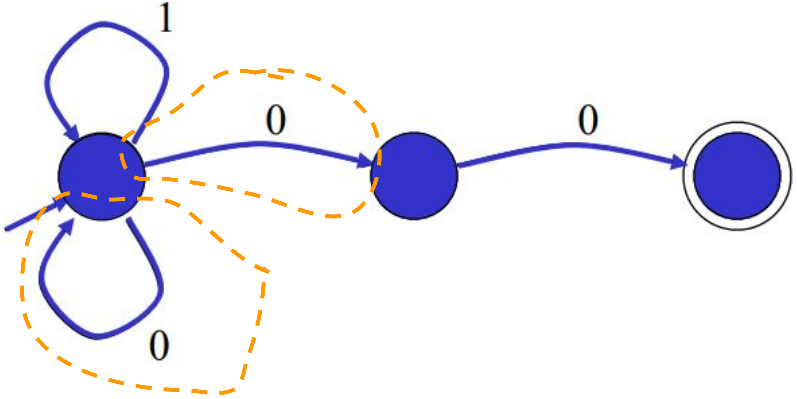
https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Nondeterministic Finite Automata(NFA)

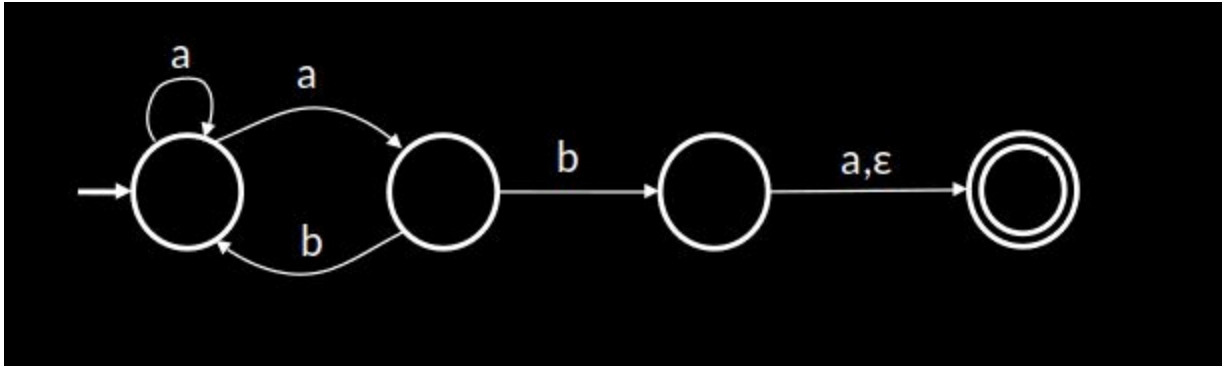
Rule: NFA accepts if it can get to a final state

- multiple paths possible (0, 1 or many at each step)
- ε-transition is a “free” move without reading input
- Accept input if some path leads to accept

Input **1 0 0**



https://web.stanford.edu/class/cs143/lectures/lecture04.pdf



Example inputs:
- ab
- aa
- aba
- abb

https://ocw.mit.edu/courses/18-404j-theory-of-computation-fall-2020/resources/mit18_404f20_lec2-1/

Examples

$L = \{s \in \{a, b\} : s \text{ starts with } a\}$

$L = \{s \in \{a, b\} : s \text{ contains } \mathbf{aa}\}$

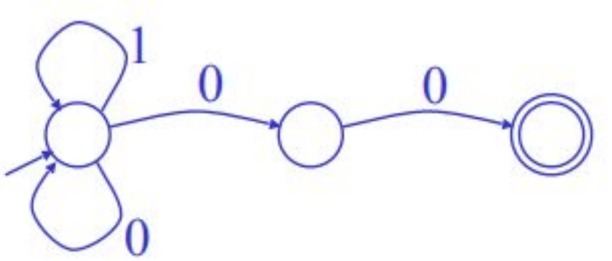
NFAs?

Regular expressions?

NFA vs DFA

- NFAs and DFAs recognize the same set of languages (regular languages)
    - For a given language NFA can be simpler than DFA
    - To determine if NFA accepts an input, we need to try all possible paths

NFA

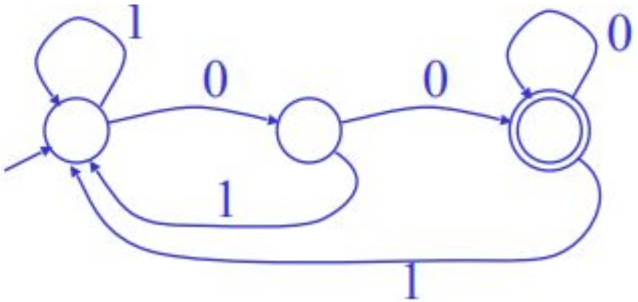


- DFAs are faster to execute

  – There are no choices to consider

- Number of transition = #states x #symbols

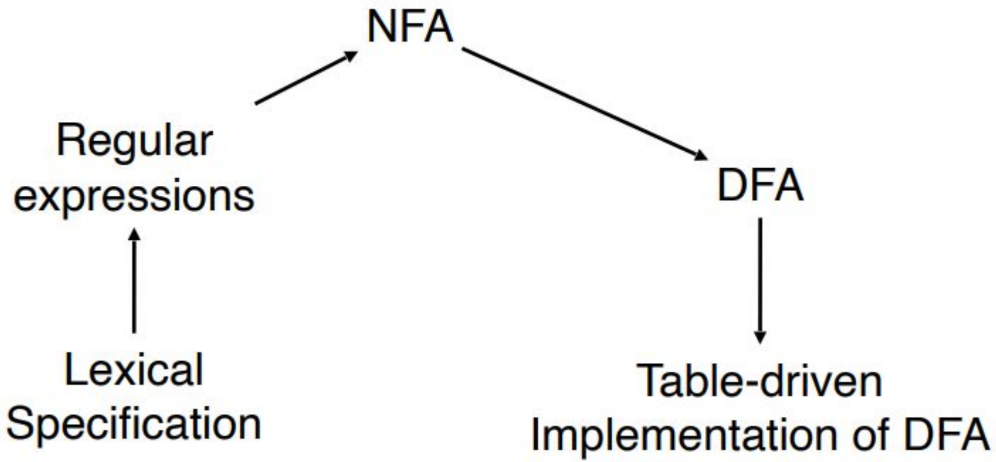
DFA



https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Convert Regular Expressions to Finite Automata

High level sketch

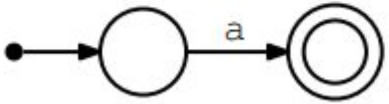


The goal is to automate the derivation of executable scanners from a collection of res.

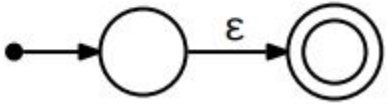
https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Convert REs to NFA(Thompson’s Construction)

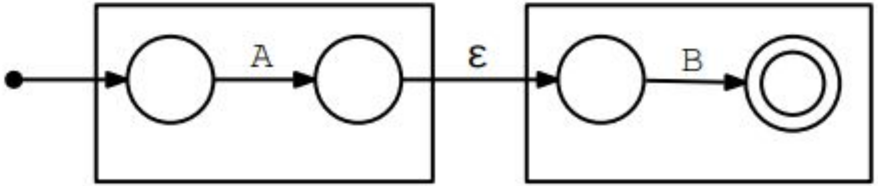
The NFA for any character a is:



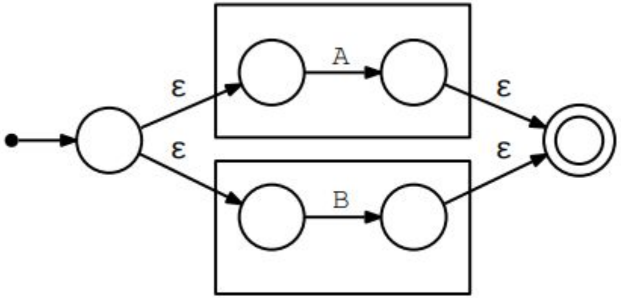
The NFA for an epsilon transition is:



The NFA for the concatenation AB is:

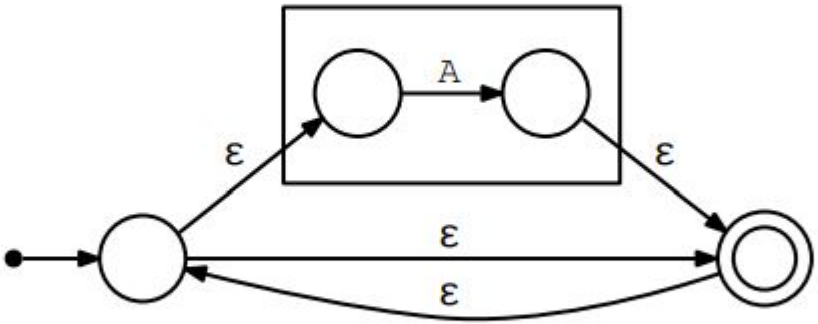


The NFA for the alternation A|B is:



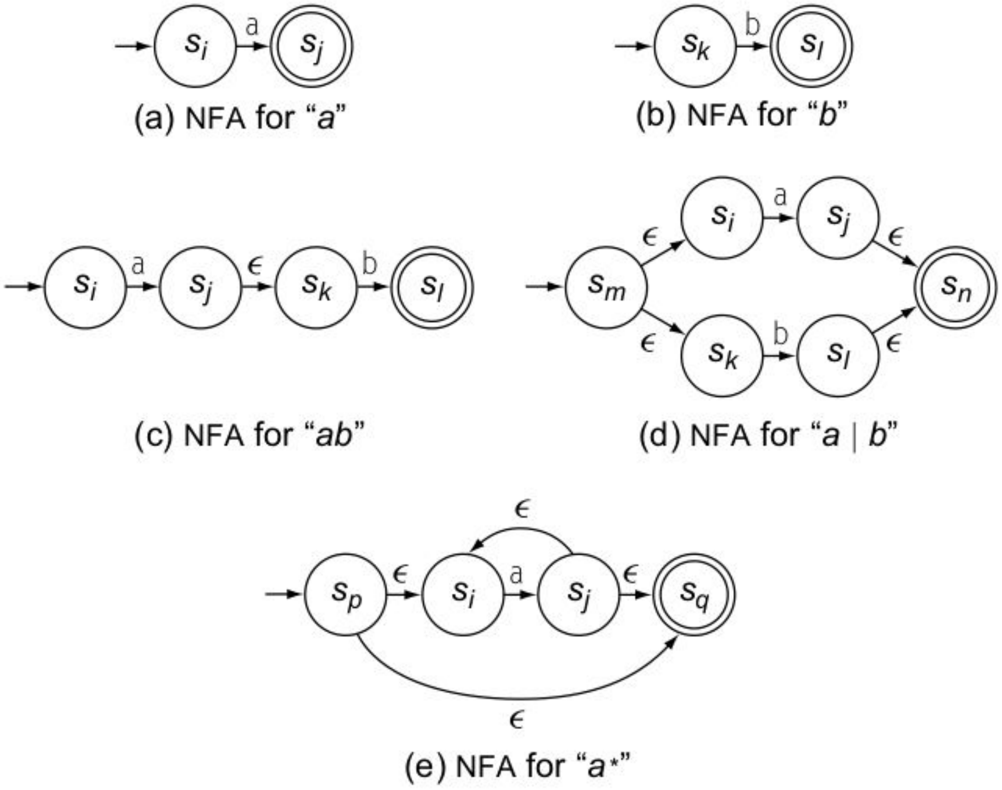
https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

The NFA for the Kleene closure A* is:



https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

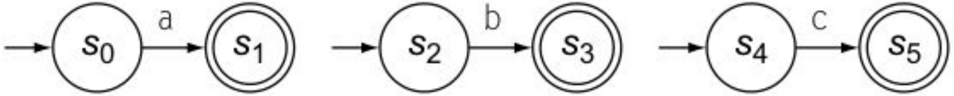
Summary figure



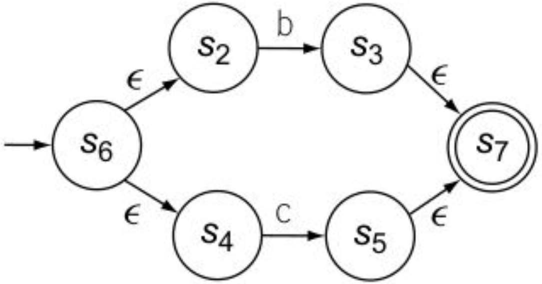
■ FIGURE 2.4 Trivial NFAs for Regular Expression Operators.

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

“a(b | c)*”

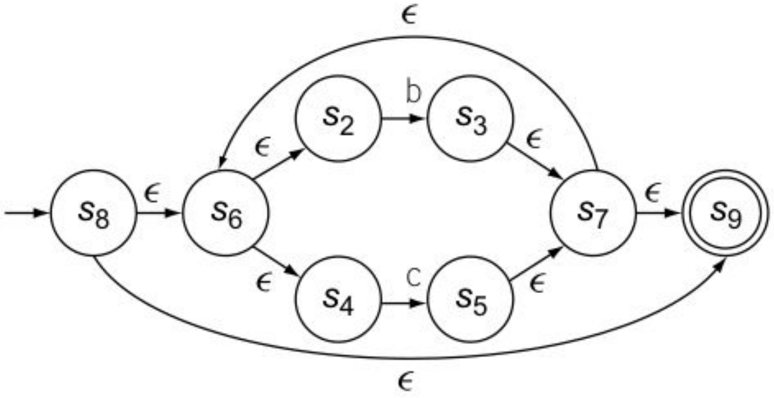


(a) NFAs for “*a*”, “*b*”, and “*c*”

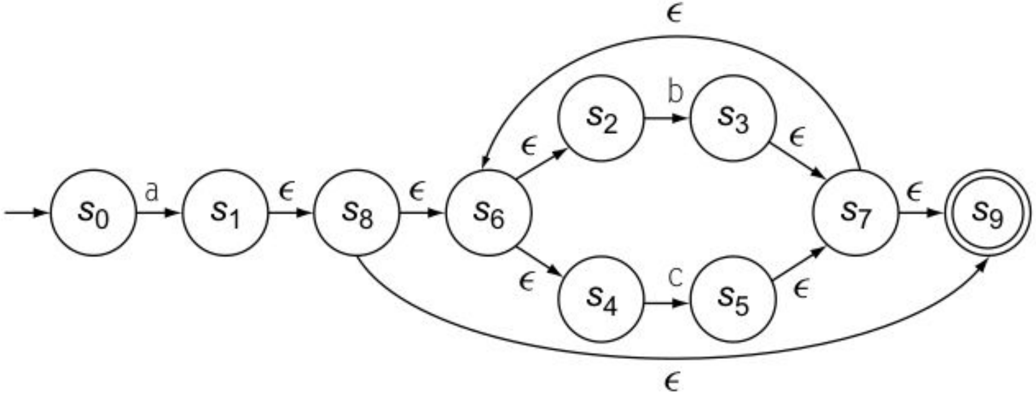


(b) NFA for “*b* | *c*”

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon



(c) NFA for "$(b \mid c)^*$"

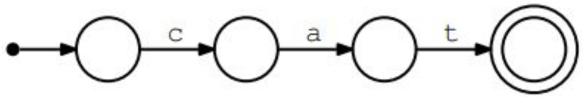


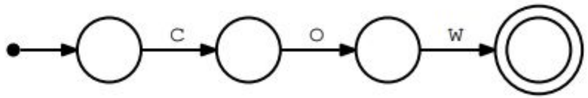
(d) NFA for "$a(b \mid c)^*$"

FIGURE 2.5 Applying Thompson's Construction to $a(b|c)^*$.

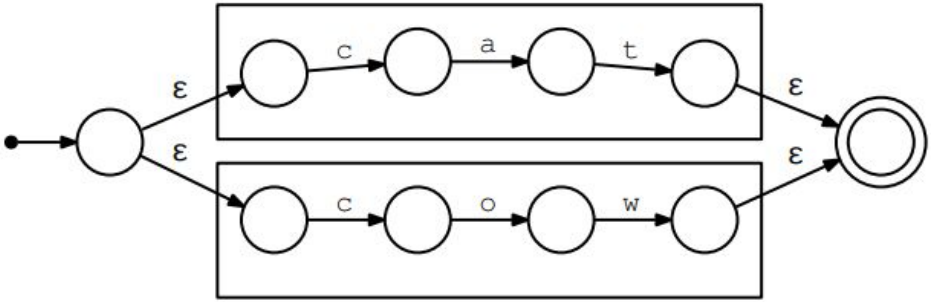
example

a(cat|cow)*



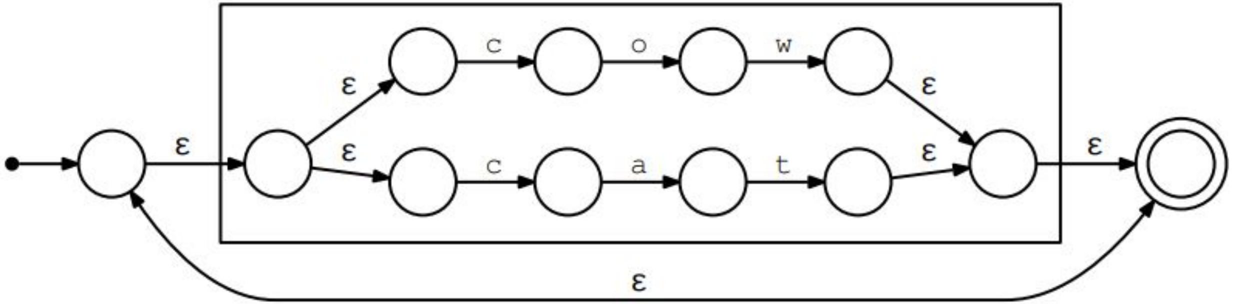


cat|cow

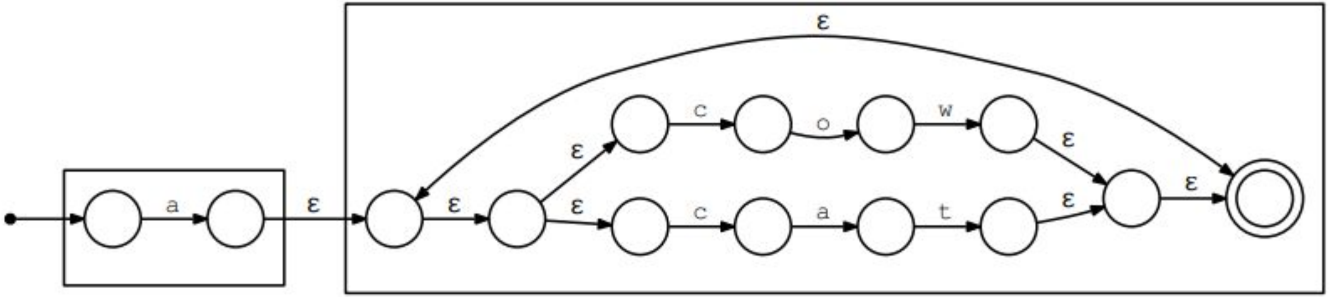


https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

Kleene closure



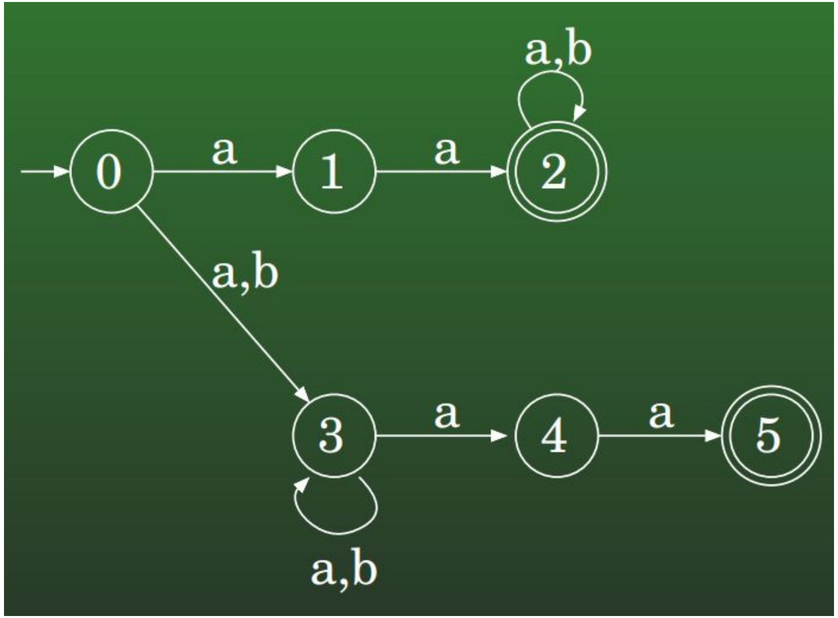
a(cat|cow)



https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

NFA vs DFA

All strings over {a,b} that begin or end with aa

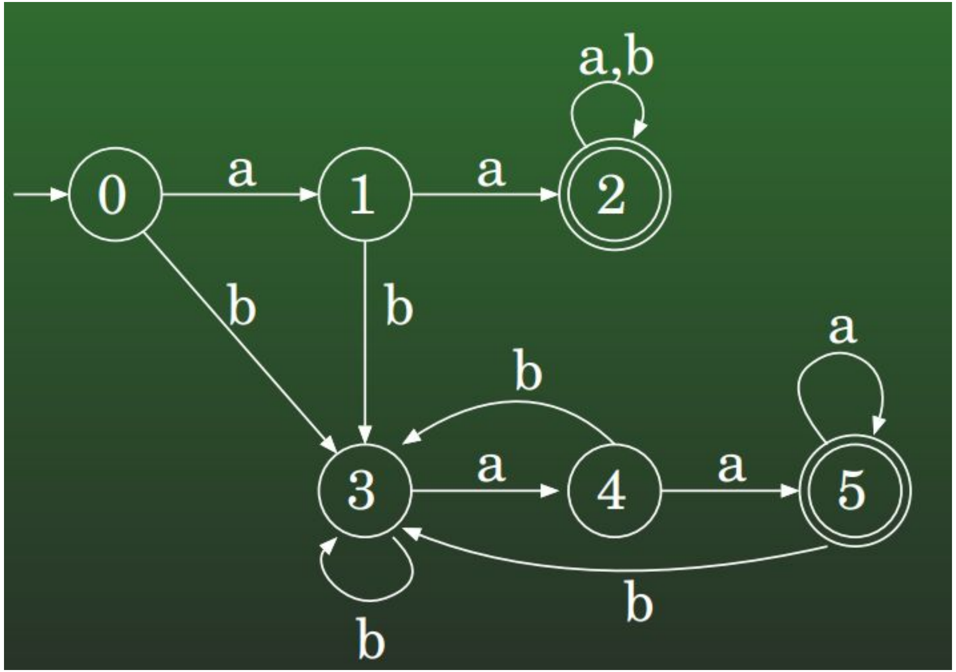


https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

Can we do the same with DFA?

NFA vs DFA

All strings over {a,b} that begin or end with aa



https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

NFA vs DFA

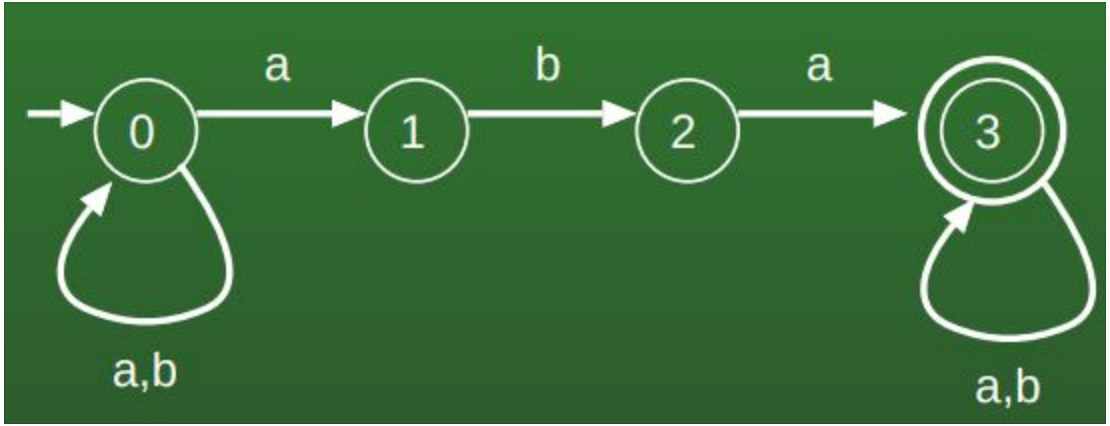
Every DFA is also an NFA

- $L_{DFA} \subseteq L_{NFA}$

- is the reverse true $L_{NFA} \subseteq L_{DFA}$?

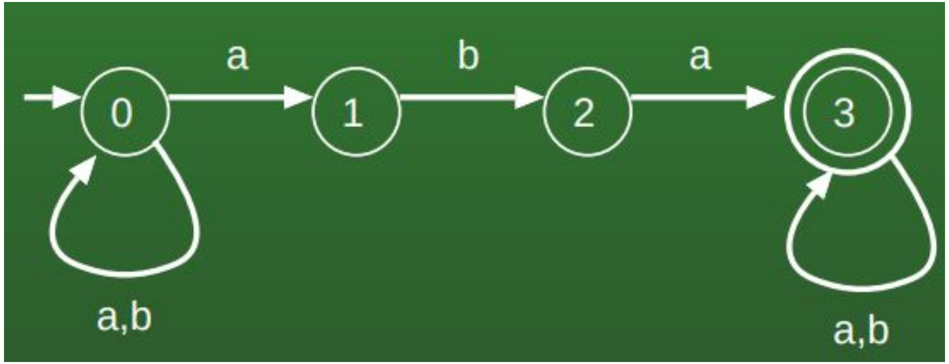
Convert NFA to DFA

NFA for all strings over {a,b} containing aba

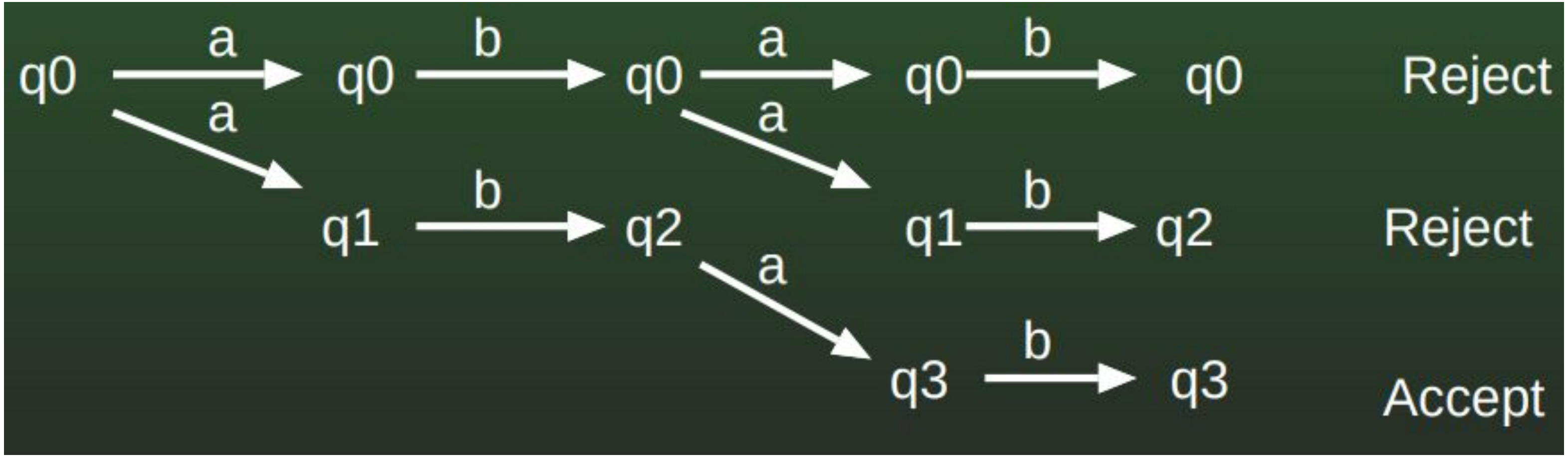


Trace **abab**

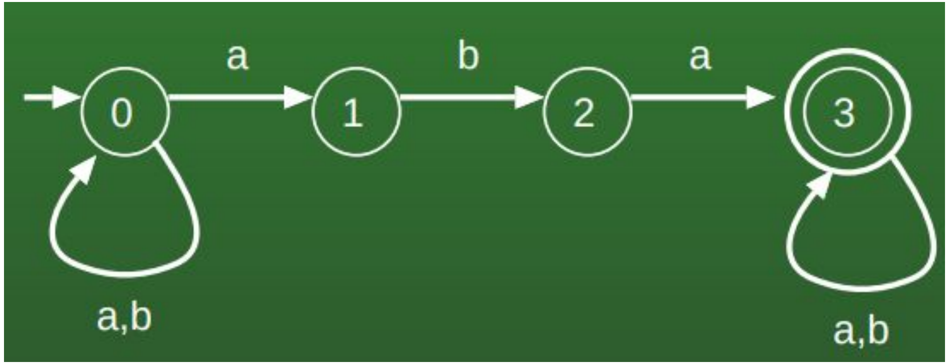
https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf



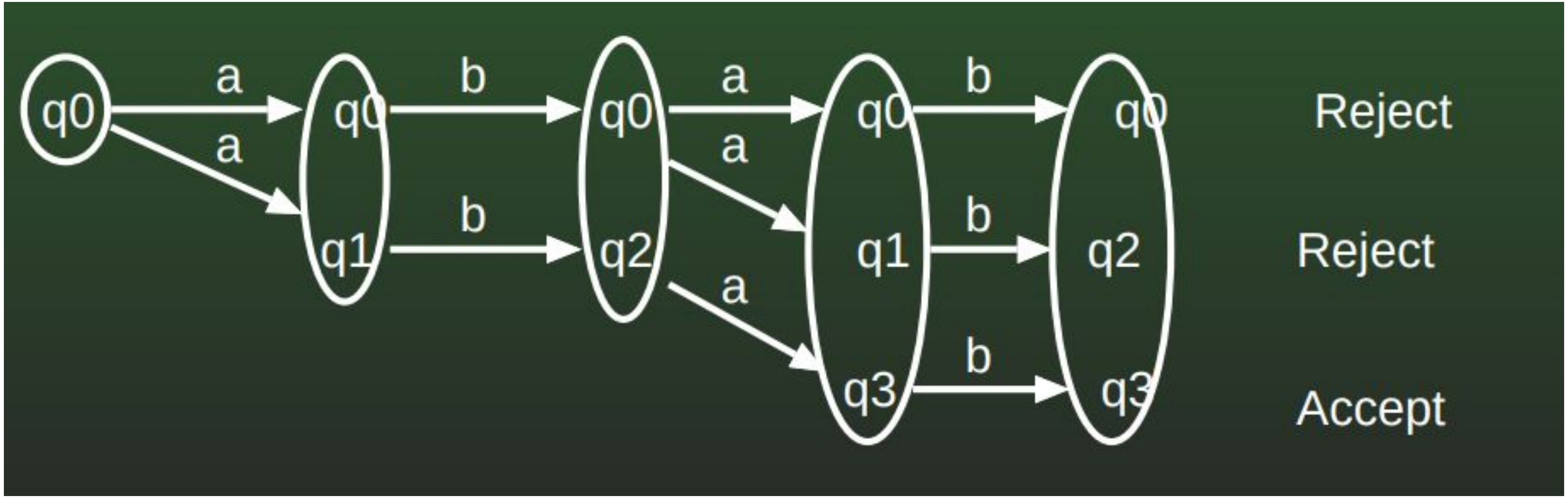
Trace **abab**



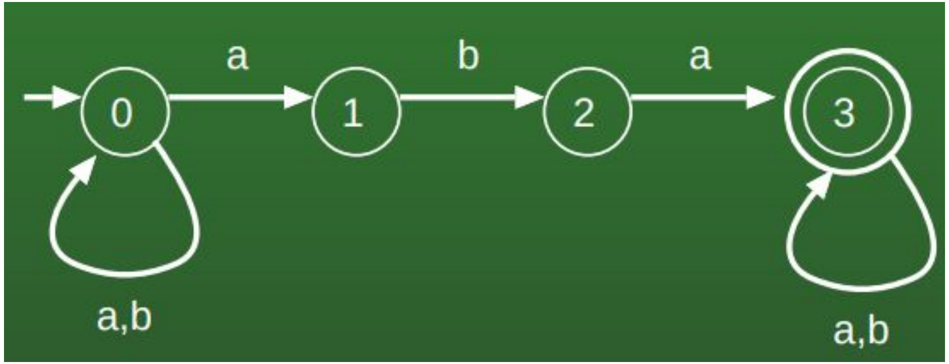
https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf



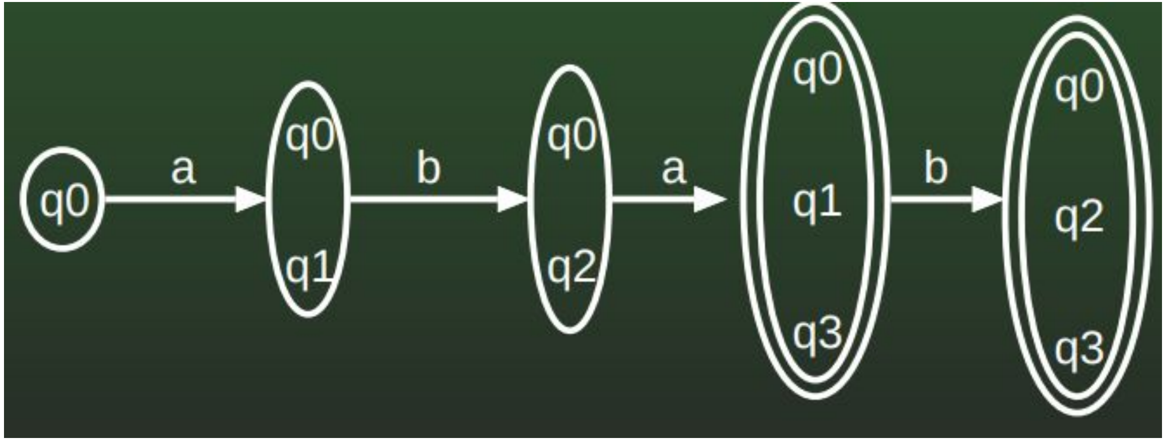
Trace **abab**



https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf



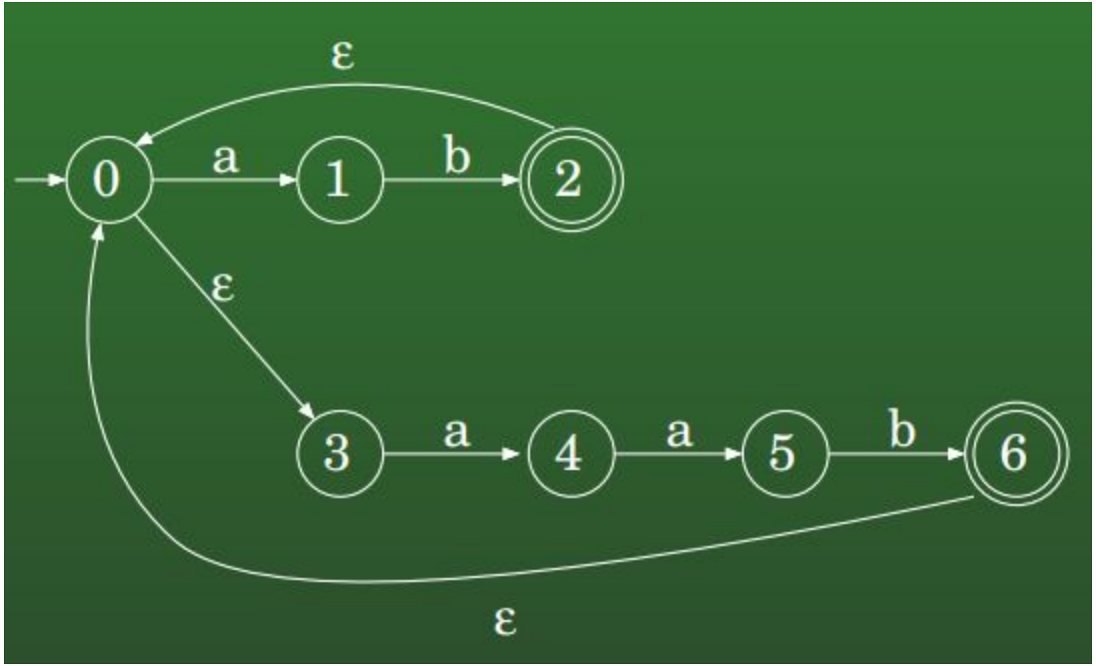
Trace **abab**



- Each state of DFA
    - = a non-empty subset of states of the NFA

https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

What about ε-moves?



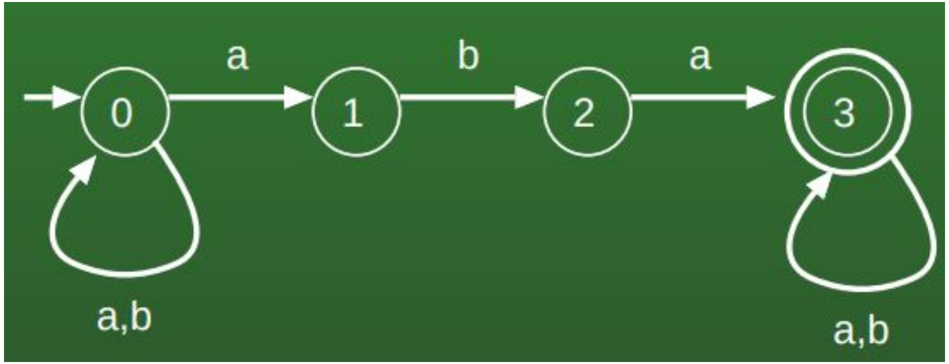
- **Start state**
  - = the set of NFA states reachable through ε-moves from NFA start state

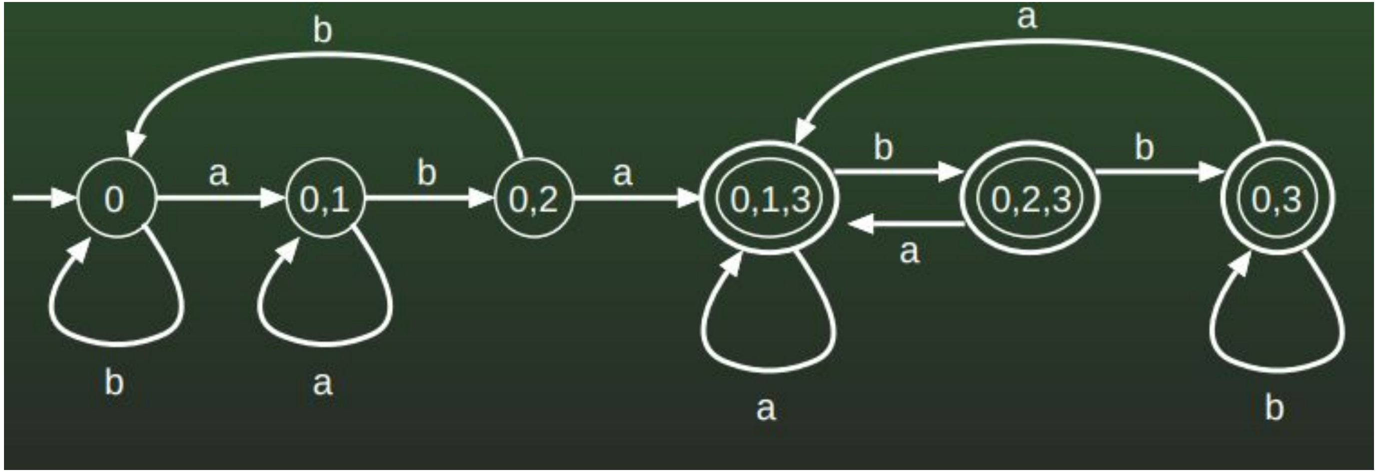
https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

NFA to DFA (general summary):

- Each state of DFA
  - = a non-empty subset of states of the NFA
- Start state
  - = the set of NFA states reachable through ε-moves from NFA start state
- Add **a** transition $S \rightarrow^{\mathbf{a}} S'$ to DFA
  - iff S' is the set of NFA states reachable from any state in S after seeing the input **a**, considering ε-moves as well

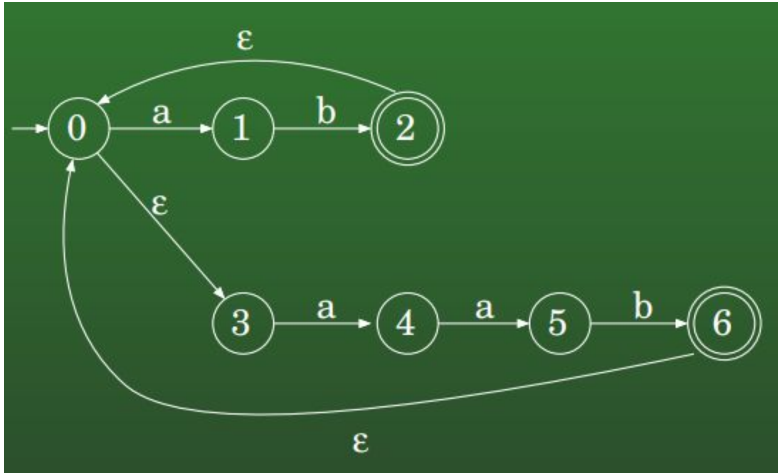
https://web.stanford.edu/class/cs143/lectures/lecture04.pdf



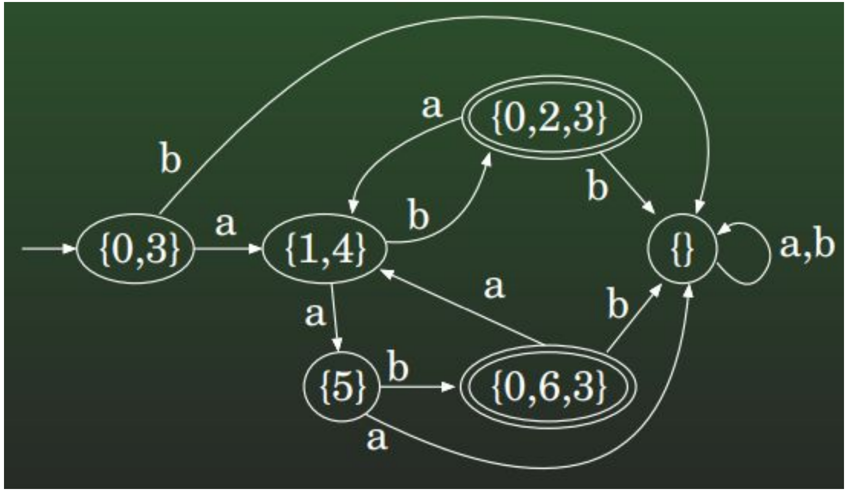


https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

NFA



DFA



https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

Remark: An NFA may be in many states at any time

- How many different states ?

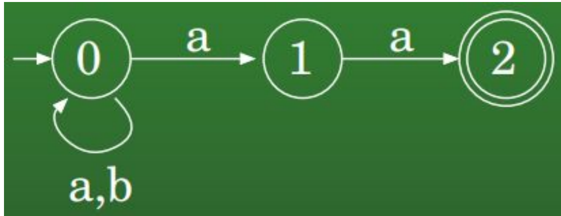
- If there are N states, the NFA must be in some subset of those N states

- How many subsets are there?

– $2^N - 1$ = finitely many

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Formal representation of states and transitions



- $K = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b\}$
- $\Delta = ((q_0, a), q_0), ((q_0, a), q_1), ((q_0, b), q_0), ((q_1, a), q_2)\}$
- $s = q_0$
- $F = \{q_2\}$

https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

Formal representation of states and transitions

- $K' = \{\{\}, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$
- $\Sigma' = \{a, b\}$
- $$\begin{aligned}\delta' = \{&((\{\}, a), \{\}), ((\{\}, b), \{\}), ((\{q_0\}, a), \{q_0, q_1\}),\\ &((\{q_0\}, b), \{q_0\}), ((\{q_1\}, a), \{q_2\}), ((\{q_1\}, b), \{\}),\\ &((\{q_2\}, a), \{\}), ((\{q_2\}, b), \{\}),\\ &((\{q_0, q_1\}, a), \{q_0, q_1, q_2\}), ((\{q_0, q_1\}, b), \{q_0\}),\\ &((\{q_0, q_2\}, a), \{q_0, q_1\}), ((\{q_0, q_2\}, b), \{q_0\}),\\ &((\{q_1, q_2\}, a), \{q_2\}), ((\{q_1, q_2\}, b), \{\}),\\ &((\{q_0, q_1, q_2\}, a), \{q_0, q_1, q_2\}), ((\{q_0, q_1, q_2\}, b), \{q_0\})\end{aligned}$$
- $s' = \{q_0\}$
- $F' = \{\{q_2\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$

https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

Proof of $L_{NFA} \subseteq L_{DFA}$

ε-closure(n) is the set of NFA states reachable from NFA state n by zero or more ε transitions.

NFA $M = (K, \Sigma, \Delta, s, F)$

DFA $M' = (K', \Sigma', \delta', s', F')$

- $K' = 2^K$
- $\Sigma' = \Sigma$
- $\delta' = \{((q_1, a), q_2) : q_1 \in K', a \in \Sigma,$
  $q_2 = \epsilon\text{-closure}\ (\{q : (q_3 \in q_1) \wedge ((q_3, a), q) \in \Delta\})$
- $s' = \epsilon\text{-closure}(s)$
- $F' = \{Q : Q \in 2^K \wedge Q \cap F \neq \emptyset\}$

https://www.cs.usfca.edu/~galles/cs411/lecture/lecture5.pdf

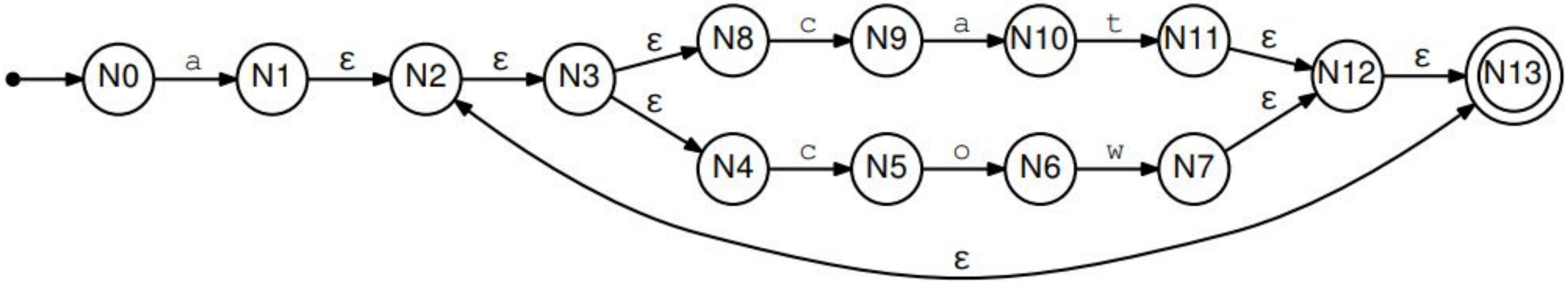
Subset construction algorithm

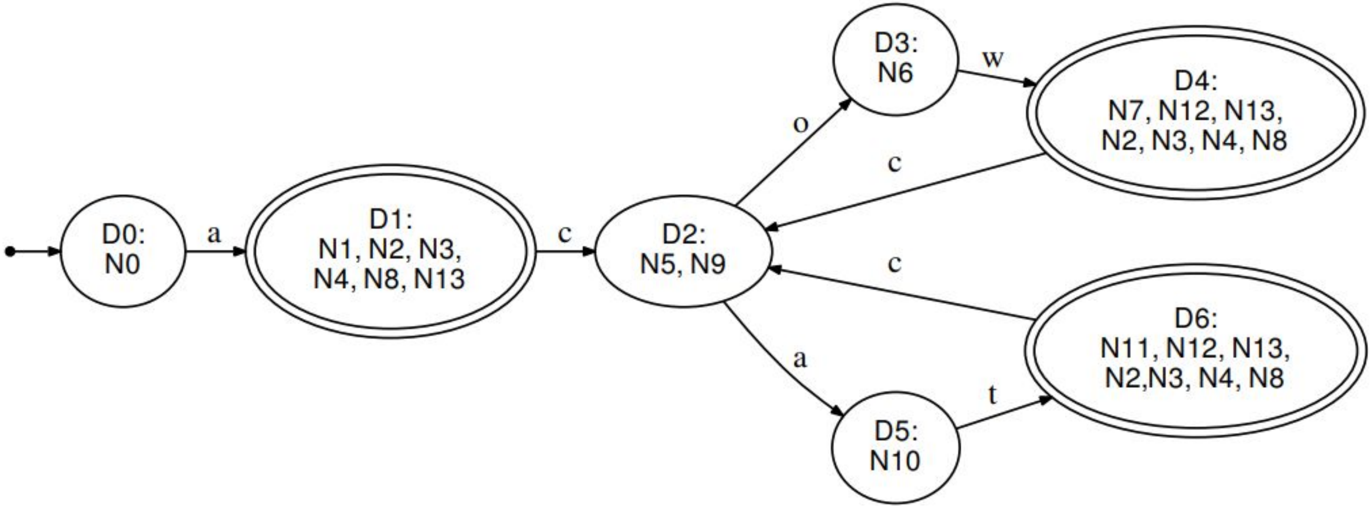
Given an NFA with states N and start state N_0, create an equivalent DFA with states D and start state D_0.

Let $D_0 = \epsilon-\text{closure}(N_0)$.
Add D_0 to a list.
While items remain on the list:
    Let d be the next DFA state removed from the list.
    For each character c in Σ:
        Let T contain all NFA states N_k such that:
            $N_j \in d$ and $N_j \xrightarrow{c} N_k$
        Create new DFA state $D_i = \epsilon-\text{closure}(\text{T})$
        If D_i is not already in the list, add it to the end.

https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

a(cat|cow)*

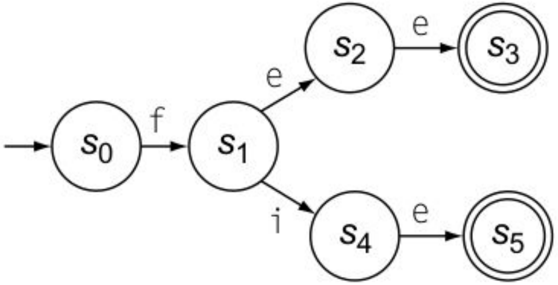




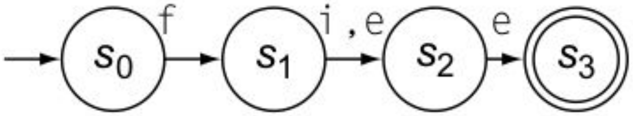
https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

DFA minimization (Hopcroft's algorithm)

Given a DFA with states S, create an equivalent DFA with an equal or fewer number of states T.



(a) DFA for "*fee* | *fie*"



(c) The Minimal DFA (States Renumbered)

The idea is to group states that behaves the same!

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

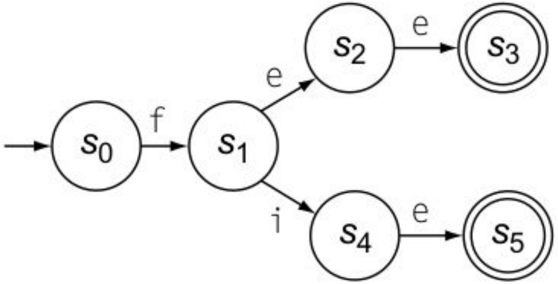
Initial group accepting non accepting states

```
T ← {D_A,  {D − D_A} };
P ← Ø

while (P ≠ T) do
    P ← T;
    T ← Ø;

    for each set p ∈ P do
        T ← T ∪ Split(p);
    end;

end;
```

```
Split(S) {
    for each c ∈ Σ do
        if c splits S into s1 and s2
            then return {s1,s2};
    end;

    return S;
}
```

■ **FIGURE 2.9** DFA Minimization Algorithm.

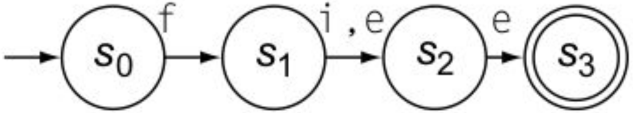
Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon



(a) 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	$\{\{s_3,s_5\},\{s_0,s_1\},\{s_2,s_4\}\}$	$\{s_0,s_1\}$	f	*split* $\{s_1\}$
4	$\{\{s_3,s_5\},\{s_0\},\{s_1\},\{s_2,s_4\}\}$	*all*	*all*	*none*

(b) Critical Steps in Minimizing the DFA



(c) The Minimal DFA (States Renumbered)

Implementation of DFA

A DFA can be implemented **by a 2D table T**

– One dimension is “states”

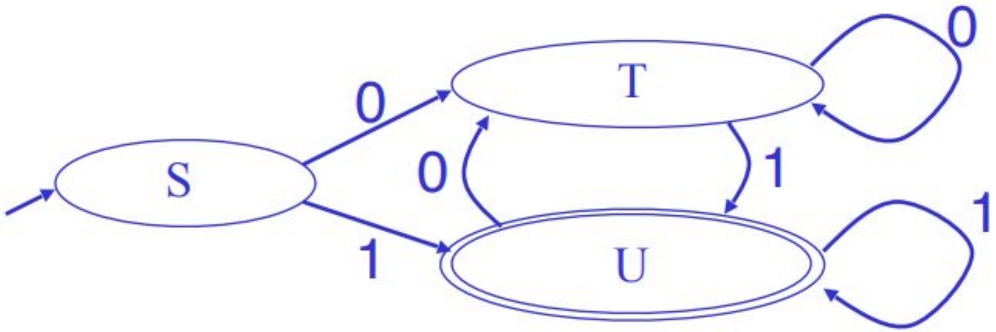
– Other dimension is “input symbol”

– For every transition $S_i \to^{a} S_k$ define T[i,a] = k

states \ input symbols	0	1
a	**a**	**b**
b	a	b
c	b	b
d	a	b

DFA “execution”
– If in state Si and input a, read T[i,a] = k and skip to state Sk
– Very efficient

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf



	0	1
S	T	U
T	T	U
U	T	U

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Using DFA as a recognizer

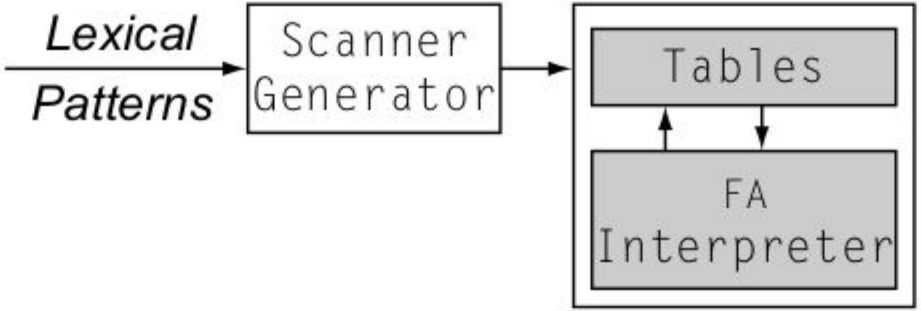
Given the REs for the various syntactic categories,

r1 , r2 , r3 , . . . , rk ,

we can construct a single re for the entire collection

(r1 | r2 | r3 | . . . | rk)

Implementing scanner



■ **FIGURE 2.13** Generating a Table-Driven Scanner.

Engineering a Compiler Second Edition Keith D. Cooper Linda Torczon

Using a Scanner(Lexer) Generator

NFA -> DFA conversion is at the heart of tools such as flex

• But, DFAs can be huge

• In practice, flex-like tools trade off speed for space in the choice of NFA and DFA representations

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Modern Lexer Tools Comparison

Beyond Flex: Lexer Tools for Different Project Paths

Tool	Language	Approach	Best For
Flex	C	Table-driven DFA	Systems-level understanding
PLY	Python	Runtime regex matching	Rapid prototyping
OCamllex	OCaml	Functional DFA	Type-safe lexers
Handwritten	Any	Direct coding	Educational understanding

Key Insight: All use the same RE → NFA → DFA theory, but with different implementations!

Lexer Implementation Patterns

```c
// State machine with explicit transitions
while((c = getchar()) != EOF) {
    switch(state) {
        case START:
            if(isdigit(c)) { state = IN_NUMBER; }
            break;
        // ... explicit state transitions
    }
  }
```

```ocaml
(* ocaml *)
let rec lex = function
  | [] -> []
  | c::cs when is_digit c ->
    let num, rest = read_number (c::cs) in
    Number(num) :: lex rest
  | c::cs when is_letter c ->
    let id, rest = read_identifier (c::cs) in
      Identifier(id) :: lex rest
```

```python
class Lexer:
    def __init__(self, text):
        self.text = text
        self.pos = 0

    def next_token(self):
        if self.pos >= len(self.text):
            return EOF
        # Pattern matching with regex
        for pattern, token_type in self.patterns:
            if match := pattern.match(self.text, self.pos):
                  return Token(token_type, match.group())
```

Using a Scanner Generator: flex

```
%{
   /* C Preamble Code */
%}
   /*Definitions*/
%%
   /*Regular Expression Rules*/
/* Rule:regex1 */    /*Actionn: {code block}*/

%%
   /*User Code*/
```

```
%{
   /* C Preamble Code */
%}
   /*Definitions*/
%%
   /*Regular Expression Rules*/
/* Rule:regex1 */    /*Action: {code block}*/
username    printf( "%s", getlogin() );

%%
   /*User Code*/
```

➔ ‘username’ is the *pattern*
➔ and the ‘printf’ is the *action*

https://westes.github.io/flex/manual/

Using a Scanner Generator

```
%{
   /* C Preamble Code */
%}
   /*Definitions*/
%%
   /*Regular Expression Rules*/
/* Rule:regex1 */    /*Action: {code block}*/
username    printf( "%s", getlogin() );

%%
   /*User Code*/
```

By default, any text not matched by a `flex` scanner is copied to the output.

The '%%' symbol marks the beginning of the rules.

```
$ flex intro.l
$ gcc lex.yy.c -ll
$ ./a.out
username
a_name
```

Another example

```
%{
int num_lines = 0, num_chars = 0;
%}
%%
\n   ++num_lines; ++num_chars;
.    ++num_chars;

%%

int main(){
   yylex();
   printf("#of lines = %d, #of chars = %d\n",
           num_lines, num_chars );
}
```

The first line declares two globals, `num_lines` and `num_chars`, which are accessible both inside `yylex()` and in the `main()` routine declared after the second ‘%%’.

The yylex() function produced by Flex uses simulated finite-state-machines (FSM) to recognize strings (or lexemes) then passes this information to the parser in the form of integer tokens.

`yylex()` is used parser to access tokens.

A credit-card example

```
/*http://web.eecs.utk.edu/~bvanderz/teac
hing/cs461Sp11/notes/flex/*/
%option noyywrap
%{
/* * * * * * * * * * * * *
* * * DEFINITIONS * * *
* * * * * * * * * * * * */
%}

%{
// recognize whether or not a credit card number is
valid
int line_num = 1;
%}

digit [0-9]
group {digit}{4}
%%
```

```
%{
/* * * * * * * * * *
* * * RULES * * *
* * * * * * * * * */
%}
  /* The carat (^) says that a credit card number must start at the
     beginning of a line and the $ says that the credit card number
     must end the line. */
^{group}([ -]?{group}){3}$  { printf(" credit card number: %s\n",
yytext); }

  /* The .* accumulates all the characters on any line that does not
     match a valid credit card number */
.* { printf("%d: error: %s \n", line_num, yytext); }
\n { line_num++; }
%%

/* * * * * * * * * * * *
* * * USER CODE * * *
* * * * * * * * * * * *
*/
int main(int argc, char *argv[]) {
 yylex();
}
```

Once the match is determined,

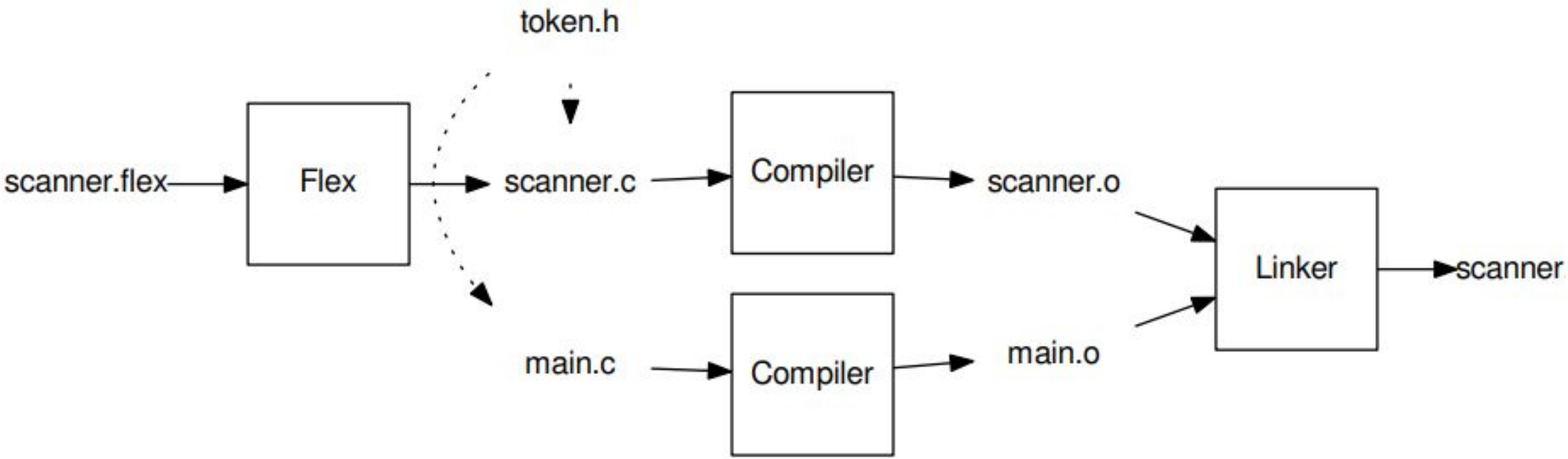
- **yytext** is the global char pointer to it.
    - its length is **yyleng**
    - the current line number is **yylineno**,

and the action corresponding to the matched pattern is then executed,

and then the remaining input is scanned for another match.

for macros and routines
See
https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Lexical.html/node12.html

You can split into multiple files



https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf

```
/*https://www3.nd.edu/~dthain/compilerbook/chap
ter3.pdf*/
%{
#include "scanner_token.h"
%}
DIGIT [0-9]
LETTER [a-zA-Z]
%%
(" "|\t|\n) /* skip whitespace */
\+          { return TOKEN_ADD; }
while       { return TOKEN_WHILE; }
{LETTER}+   { return TOKEN_IDENT; }
{DIGIT}+    { return TOKEN_NUMBER; }
.           { return TOKEN_ERROR; }
%%
int yywrap() { return 1; }
```

```
/*scanner_token.h
https://www3.nd.edu/~dthain/compilerbook/chapte
r3.pdf*/
typedef enum {
   TOKEN_EOF=0,
   TOKEN_WHILE,
   TOKEN_ADD,
   TOKEN_IDENT,
   TOKEN_NUMBER,
   TOKEN_ERROR
} token_t;
```

Some practical considerations

Ambiguity

```
/* Problem: 'if' matches both rules */
if      { return KEYWORD_IF; }
[a-z]+  { return IDENTIFIER; }

  /* Solution: Order matters! Put keywords first */
```

Performance Considerations:

- DFA: Fast but large memory footprint
- NFA: Smaller but backtracking overhead
- Trade-off: Most tools (like Flex) use optimized DFA

Error Recovery Strategies:

1. Skip invalid character and continue
2. Report line numbers for debugging
3. Try to find next valid token boundary

Different Languages, Different Challenges

JavaScript (Automatic Semicolon Insertion):

```
return     // ASI inserts semicolon here!
      { result: 42 };    // This becomes unreachable!
```

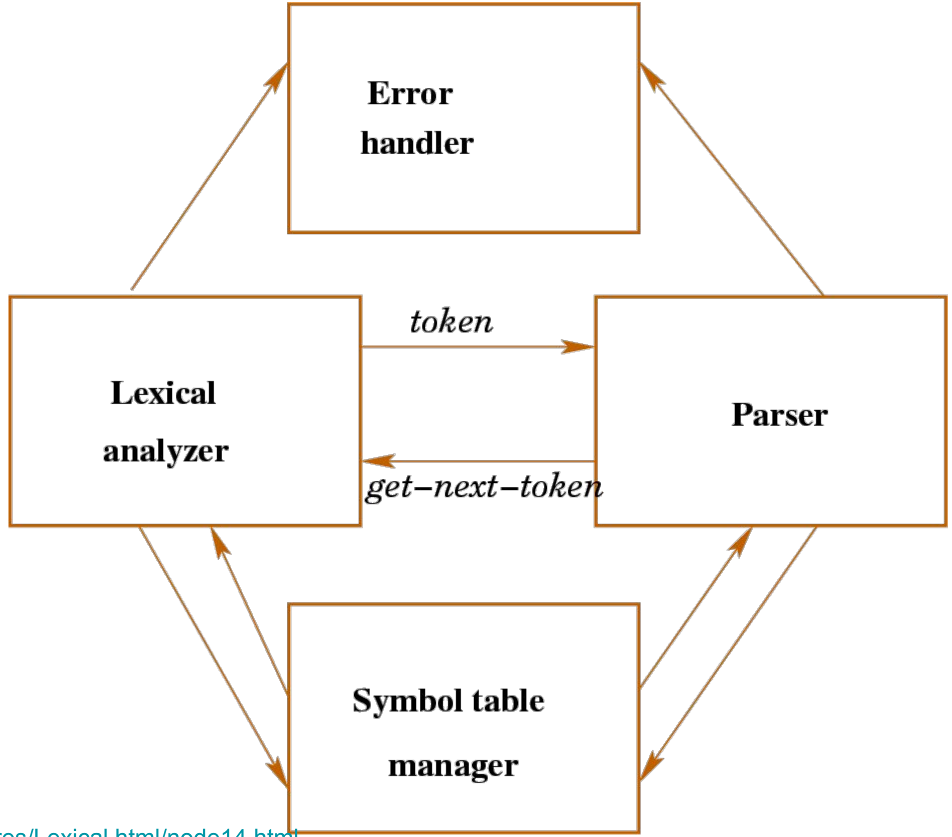
Python (Significant Whitespace):

```
# Indentation matters!
def hello():
    print("indented")  # T_INDENT token
  print("dedented")      # T_DEDENT token
```

SQL (Context-Sensitive Keywords):

```
-- 'value' is keyword here:
CREATE TABLE test (id INT, value VARCHAR(20))

-- 'value' is identifier here:
  SELECT value FROM test WHERE id = 1
```

Next week: parser



https://www.csd.uwo.ca/~mmorenom/CS447/Lectures/Lexical.html/node14.html

```c
/*
https://www3.nd.edu/~dthain/compilerbook/chapter3.pdf
*/
#include "scanner_token.h"
#include <stdio.h>
extern FILE *yyin;
extern int yylex();
extern char *yytext;
int main()
{
    yyin = fopen("program.c", "r");
    if (!yyin) {
        printf("could not open program.c! \n");
        return 1;
    }
    while (1){
        token_t t = yylex();
        if (t == TOKEN_EOF)
            break;
        printf("token: %d text: %s\n", t, yytext);
    }
}
```

Project-1: Specifying a PL (Simple Demos)

- Specify rules for identifiers, keywords, etc. for a programming language
- Write regular expressions for each of them
- Use flex or similar to tokenize a given input file

Project 1: Example mini language

MiniLang Lexer Implementation Tips

All Paths must recognize some form of:

- Numbers: `123`, `45.67`
- Identifiers: `variable`, `calculate_sum`
- Keywords: `let`, `def`, `if`, `else`
- Operators: `+`, `-`, `*`, `/`
- Punctuation: `(`, `)`, `=`, `,`

C/Flex Path:

```
DIGIT    [0-9]
LETTER   [a-zA-Z]
%%
"let"    { return LET; }
{LETTER}({LETTER}|{DIGIT})* { return IDENTIFIER; }
{DIGIT}+ { return NUMBER; }
```

Python/PLY Path:

```python
tokens = ('LET', 'IDENTIFIER', 'NUMBER')

def t_LET(t):
    r'let'
    return t

def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z_0-9]*'
    return t
```

OCaml/LLVM Path:

```ocaml
rule token = parse
| "let" { LET }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id
    { IDENTIFIER id }
| ['0'-'9']+ as num { NUMBER (int_of_string num) }
```

Testing Your Lexer

Basic Recognition:

```
// Should tokenize as: LET ID("x") EQ NUMBER(5)
  let x = 5
```

Edge Cases:

```
// Maximum munch: "identifier" not "ident if ier"
identifier

// Keyword vs identifier: "if" vs "ifcondition"
if        // KEYWORD_IF
ifcondition  // IDENTIFIER
```

Error Handling:

```
// Should skip @ and continue
let x@y = 5  // LET ID("x") ID("y") EQ NUMBER(5)
```

From Lexing to Parsing

```
Source Code
    → [Lexer] → Token Stream
    → [Parser] → Abstract Syntax Tree
```

```
# Source: "let result = 10 + 20"
  [LET, ID("result"), EQ, NUMBER(10), PLUS, NUMBER(20), EOF]

// Simple lexer API
token_t next_token();
const char* token_text();
int token_line();
```

Next Week Preview: The parser will use your tokens to build the program structure!

Examples: minilang.l

```
%{
#include <stdio.h>
#include "tokens.h"

int line_num = 1;
%}

DIGIT    [0-9]
LETTER   [a-zA-Z_]
WS       [ \t]

%%
\n       { line_num++; }
{WS}+    /* skip whitespace */

"let"    { printf("LINE %d: KEYWORD 'let'\n", line_num); return LET; }
"def"    { printf("LINE %d: KEYWORD 'def'\n", line_num); return DEF; }
"+"      { printf("LINE %d: OPERATOR '+'\n", line_num); return PLUS; }
"="      { printf("LINE %d: OPERATOR '='\n", line_num); return EQUALS;
}
```

```
{LETTER}({LETTER}|{DIGIT})* {
    printf("LINE %d: IDENTIFIER '%s'\n", line_num, yytext);
    return IDENTIFIER;
}

{DIGIT}+ {
    printf("LINE %d: NUMBER '%s'\n", line_num, yytext);
    return NUMBER;
}

.        { printf("LINE %d: ERROR: unexpected '%s'\n", line_num, yytext); }
%%

  int yywrap() { return 1; }
```

tokens.h

```c
#ifndef TOKENS_H
#define TOKENS_H

typedef enum {
    LET = 258,
    DEF,
    IDENTIFIER,
    NUMBER,
    PLUS,
    EQUALS,
    END_OF_FILE
} token_type;

#endif
```

```
# In terminal:
flex minilang.l # this gives lex.yy.c
gcc lex.yy.c test.c -o lexer
./lexer
```

test.c

```c
#include <stdio.h>
#include "tokens.h"

extern FILE *yyin;
extern int yylex();
extern char *yytext;

int main() {
    // Test with simple input
    yyin = fopen("test_input.txt", "w");
    fprintf(yyin, "let x = 5\n");
    fprintf(yyin, "def add a b = a + b");
    fclose(yyin);

    yyin = fopen("test_input.txt", "r");

    int token;
    while ((token = yylex()) != 0) {
        printf("Token: %d, Text: %s\n", token, yytext);
    }

    fclose(yyin);
    return 0;
  }
```

Python PLY lexer

```python
import ply.lex as lex

# Token list
tokens = (
    'LET', 'DEF', 'IDENTIFIER', 'NUMBER',
    'PLUS', 'EQUALS'
)

# Simple tokens
t_PLUS = r'\+'
t_EQUALS = r'='
t_ignore = ' \t'

def t_LET(t):
    r'let'
    print(f"KEYWORD 'let' at line {t.lineno}")
    return t

def t_DEF(t):
    r'def'
    print(f"KEYWORD 'def' at line {t.lineno}")
    return t
```

```python
def t_IDENTIFIER(t):
    r'[a-zA-Z_][a-zA-Z0-9_]*'
    print(f"IDENTIFIER '{t.value}' at line {t.lineno}")
    return t

def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    print(f"NUMBER {t.value} at line {t.lineno}")
    return t

def t_newline(t):
    r'\n+'
    t.lexer.lineno += len(t.value)

def t_error(t):
    print(f"Illegal character '{t.value[0]}' at line {t.lineno}")
    t.lexer.skip(1)
```

```python
# Build the lexer
lexer = lex.lex()

# Test it
data = '''
let x = 5
def add a b = a + b
'''

print("=== PYTHON LEXER DEMO ===")
lexer.input(data)
for token in lexer:
      print(f"Token: {token.type}, Value: {token.value}")
```

```
$ python minilang_ply.py
```

Lexer and parser generators (ocamllex, ocamlyacc)

minilang.mll

```
{
  open Printf

  type token =
    | LET
    | DEF
    | IDENTIFIER of string
    | NUMBER of int
    | PLUS
    | EQUALS
    | EOF
    | ERROR of char

  let string_of_token = function
    | LET -> "LET"
    | DEF -> "DEF"
    | IDENTIFIER s -> sprintf "IDENTIFIER(%s)" s
    | NUMBER n -> sprintf "NUMBER(%d)" n
    | PLUS -> "PLUS"
    | EQUALS -> "EQUALS"
    | EOF -> "EOF"
    | ERROR c -> sprintf "ERROR('%c')" c
}
```

```
rule token = parse
| [' ' '\t']     { token lexbuf }
| '\n'           { Lexing.new_line lexbuf; token lexbuf }
| "let"          { printf "KEYWORD 'let' at line %d\n" (Lexing.lexeme_start lexbuf |> fst);
LET }
| "def"          { printf "KEYWORD 'def' at line %d\n" (Lexing.lexeme_start lexbuf |> fst);
DEF }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id
    { printf "IDENTIFIER '%s' at line %d\n" id (Lexing.lexeme_start lexbuf |> fst);
      IDENTIFIER id }
| ['0'-'9']+ as num
    { printf "NUMBER %s at line %d\n" num (Lexing.lexeme_start lexbuf |> fst);
      NUMBER (int_of_string num) }
| '+'            { printf "OPERATOR '+' at line %d\n" (Lexing.lexeme_start lexbuf |> fst);
PLUS }
| '='            { printf "OPERATOR '=' at line %d\n" (Lexing.lexeme_start lexbuf |> fst);
EQUALS }
| eof            { EOF }
  | _ as c         { printf "ERROR: unexpected '%c' at line %d\n" c (Lexing.lexeme_start

  lexbuf |> fst); ERROR c }
```

test_lexer.ml

```ocaml
let () =
  let test_input = "let x = 5\ndef add a b = a + b" in
  let lexbuf = Lexing.from_string test_input in

  print_endline "=== OCAML LEXER DEMO ===";

  let rec print_tokens lexbuf =
    match Minilang.token lexbuf with
    | Minilang.EOF ->
        print_endline "End of input"
    | token ->
        print_endline (Minilang.string_of_token token);
        print_tokens lexbuf
  in

    print_tokens lexbuf
```

test_input.txt

```
let x = 42
def add a b = a + b
let result = add x 10
@invalid character
```

```
# In terminal:
ocamllex minilang.mll
ocamlc -c minilang.ml
ocamlc -c test_lexer.ml
ocamlc -o test_lexer minilang.cmo test_lexer.cmo
./test_lexer
```

Tips on Building Large Systems

- KISS (Keep It Simple, Stupid!)
- Don’t optimize prematurely
- Design systems that can be tested
- It is easier to modify a working system than to get a system working

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

Value simplicity

“It's not easy to write good software. […] it has a lot to do with valuing simplicity over complexity.” - Barbara Liskov

“Debugging is twice as hard as writing the code in the first place.Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.” - Brian Kernighan

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.” - Tony Hoare

“Simplicity does not precede complexity, but follows it.” - Alan Perlis

https://web.stanford.edu/class/cs143/lectures/lecture04.pdf

hw2

Written assignment on FAs, NFAs