Compilation 2024

Lexical Analysis

Amin Timany
timany@cs.au.dk

Revised from slides by Aslan Askarov, E. Ernst, J. Midtgaard, and M. I. Schwartzbach

Where do ASTs come from?

- So far we have worked with ASTs as input to our compilers
- Programmers write source code (sequence of characters).
- Where do ASTs come from?

Lexical Analysis & Parsing

Lexical analysis

Lexical analysis

Lexical analysis

First phase in the compilation

Lexical analysis

First phase in the compilation

Input: stream of characters

Lexical analysis

First phase in the compilation

Input: stream of characters

i	f		(	x	>	0	)	\n	\t	t	h	e	n		1	\n	\t	e	l	s	e		0

Lexical analysis

First phase in the compilation

Input: stream of characters

i	f		(	x	>	0	)	\n	\t	t	h	e	n		1	\n	\t	e	l	s	e		0

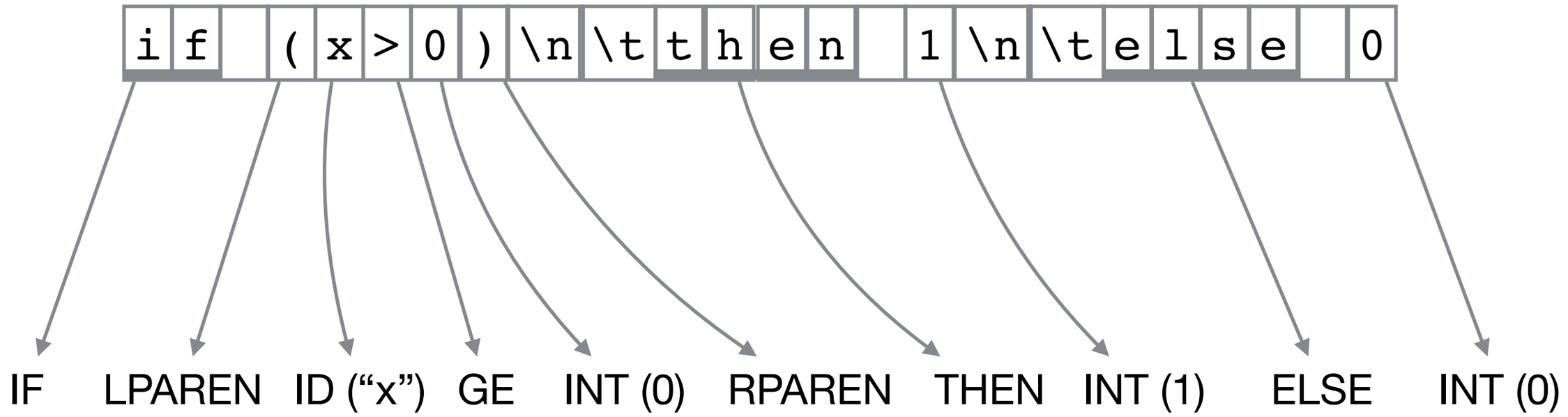
IF LPAREN ID (“x”) GE INT (0) RPAREN THEN INT (1) ELSE INT (0)

Output: stream of *tokens* in our language

Lexical analysis

First phase in the compilation

Input: stream of characters

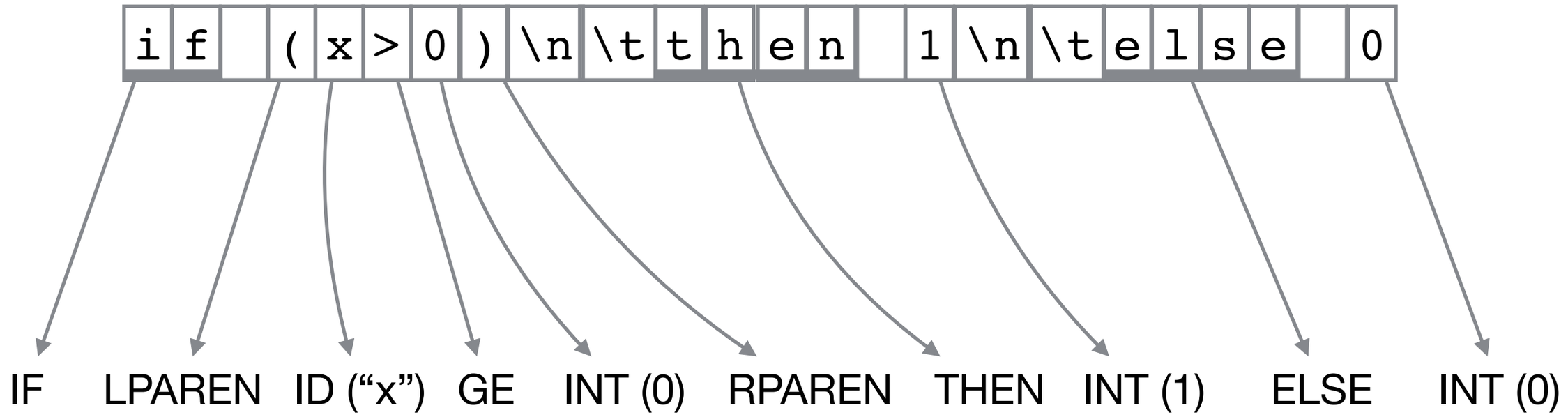


Output: stream of *tokens* in our language

Lexical analysis

First phase in the compilation

Input: stream of characters



Output: stream of *tokens* in our language

Discards comments, whitespace, newline, tab characters, preprocessor directives

Tokens

Type	Examples
ID	`foo  n14  a' my-fun`
INT	`73 0 070`
REAL	`0.0 .5 10.`
IF	`if`
COMMA	`,`
LPAREN	`(`
ASGMT	`:=`

Non-tokens

Type	Examples
comments	`/* dead code */`
	`// comment`
	`(* nest (*ed*) *)`
preprocessor directives	`#define N 10`
	`#include <stdio.h>`
whitespace	

Token data structure

- Many tokens need no associated data, e.g.:
  IF, COMMA, LPAREN, RPAREN, ASGMT

- Some tokens carry an associated string:
  ID (“my-fun”)

- Some tokens carry associated data of other types:
  INT (73), INT (1), FLOAT (IEEE754, 1001111100…)

- Tokens may include useful additional information:
  start/end pos in input file (line number + column, or charpos)

Q: How many *token types* are there in this program?

```
var x = 0;
x = x+1; /*add one*/
print_integer(x);
```

Q/A

- Consider source program

  var 😀 := 0.0

- Language: case sensitive, ASCII

- How to report error of using 😀?

Q/A

- Consider source program

  var 😀 := 0.0

- Language: case sensitive, ASCII

- How to report error of using 😀?

  `FileName:Line.Col: Illegal character` 😀

Regular expressions

- We can use *regular expressions* to specify programming language tokens
- Regular expressions R
  - Expected to be well-known (4th semester)
  - Syntax
    - character a
    - choice $R_1 \mid R_2$
    - concat $R_1 \cdot R_2$ also sometimes $R_1 R_2$
    - empty string ε
    - repeat $R*$

Regular expressions used for scanning

Examples

```
if                                                    {IF};
[a-z][a-z0-9]*                                        {ID};
[0-9]+                                                {NUM};
([0-9]+"."[0-9]*) | ([0-9]* "." [0-9]+)               {REAL};
("--" [a-z]*"\n") | (" "|"\t")                        { continue()};
_                                                     { error();continue()};
```

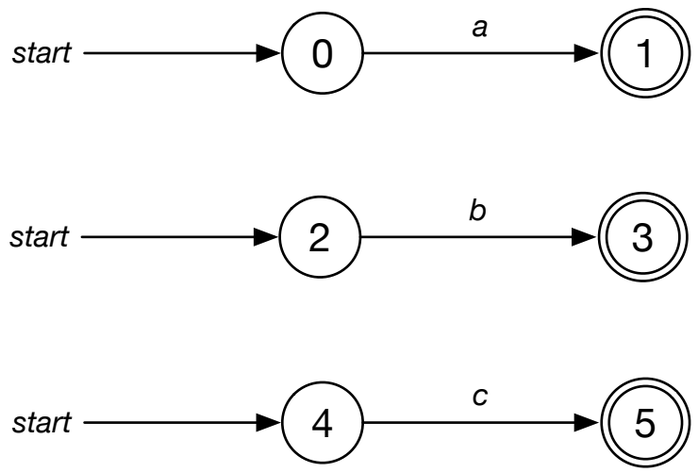
OBS: exact syntax is different from tool to tool

From regular expressions to DFA (recap)

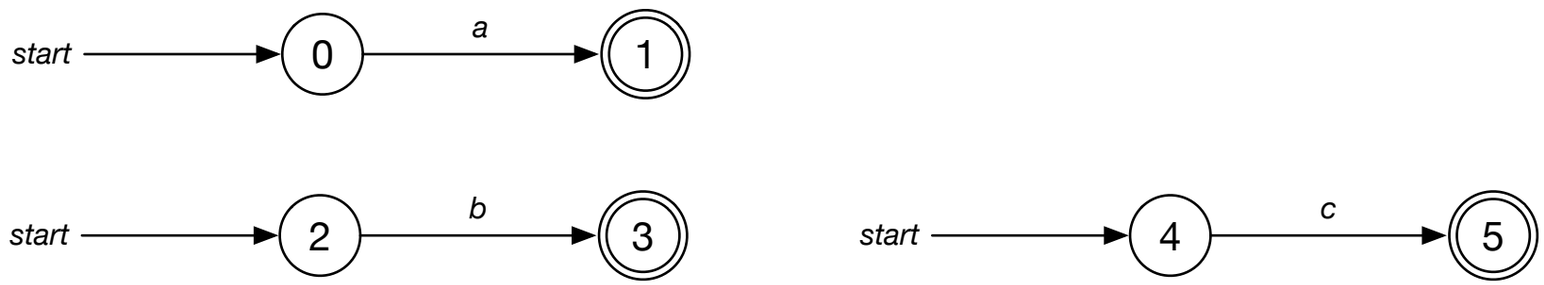
1. Regexp to NFA (Thompson's Construction)
2. NFA to DFA (The Subset Construction)
3. DFA minimization: (Hopcroft's algorithm)

Example $(a \mid bc)^*$

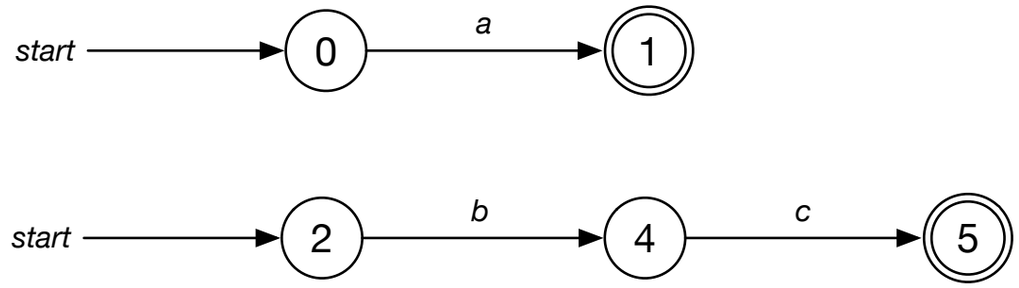
From regexp to NFA (a|bc)*



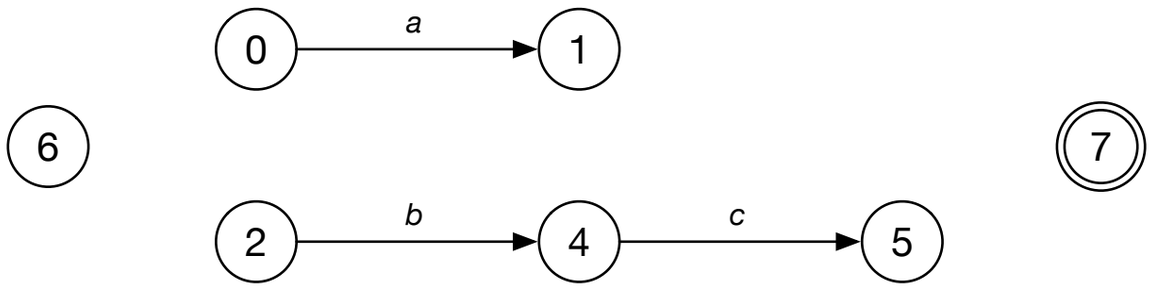
From regexp to NFA (a|bc)*



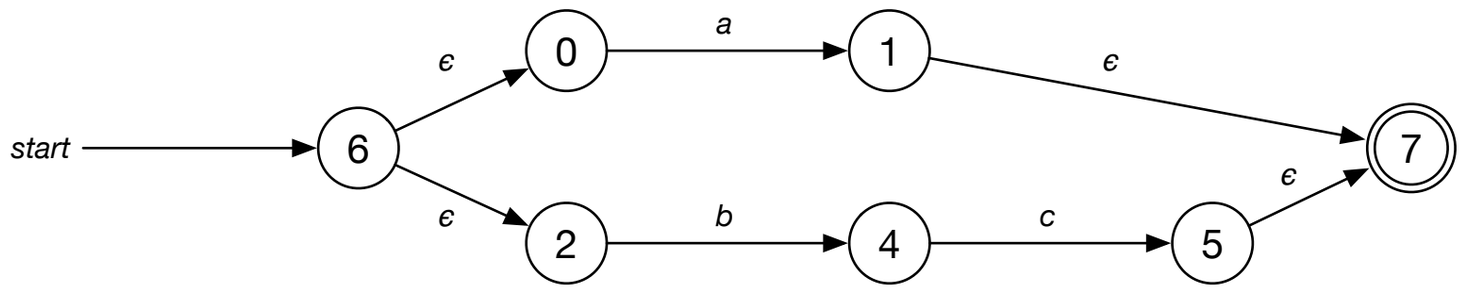
From regexp to NFA (a|bc)*



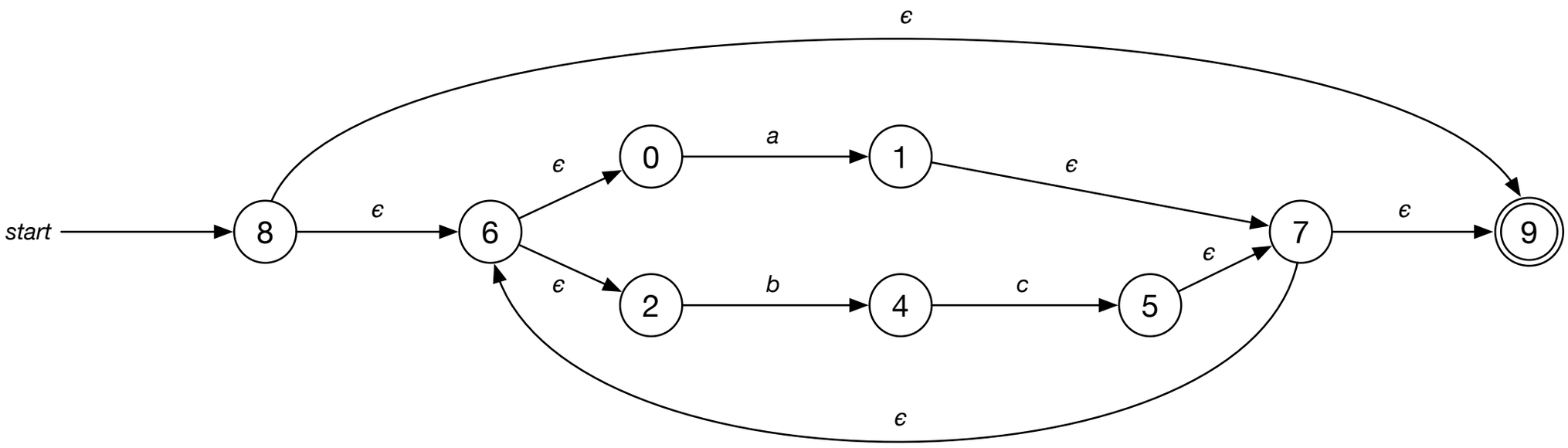
From regexp to NFA (a|bc)*



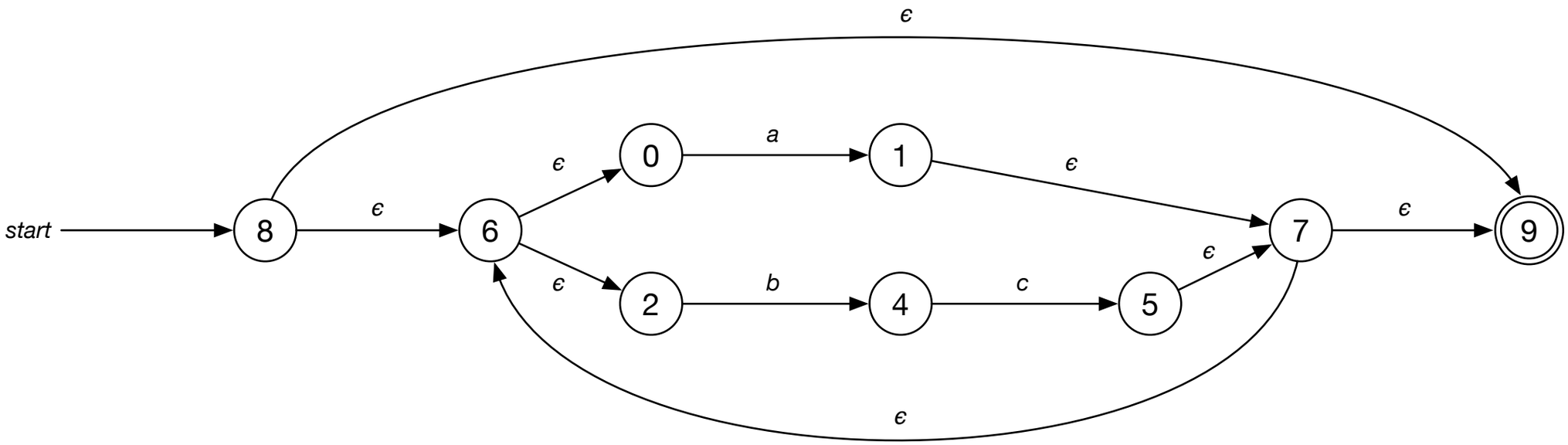
From regexp to NFA (a|bc)*

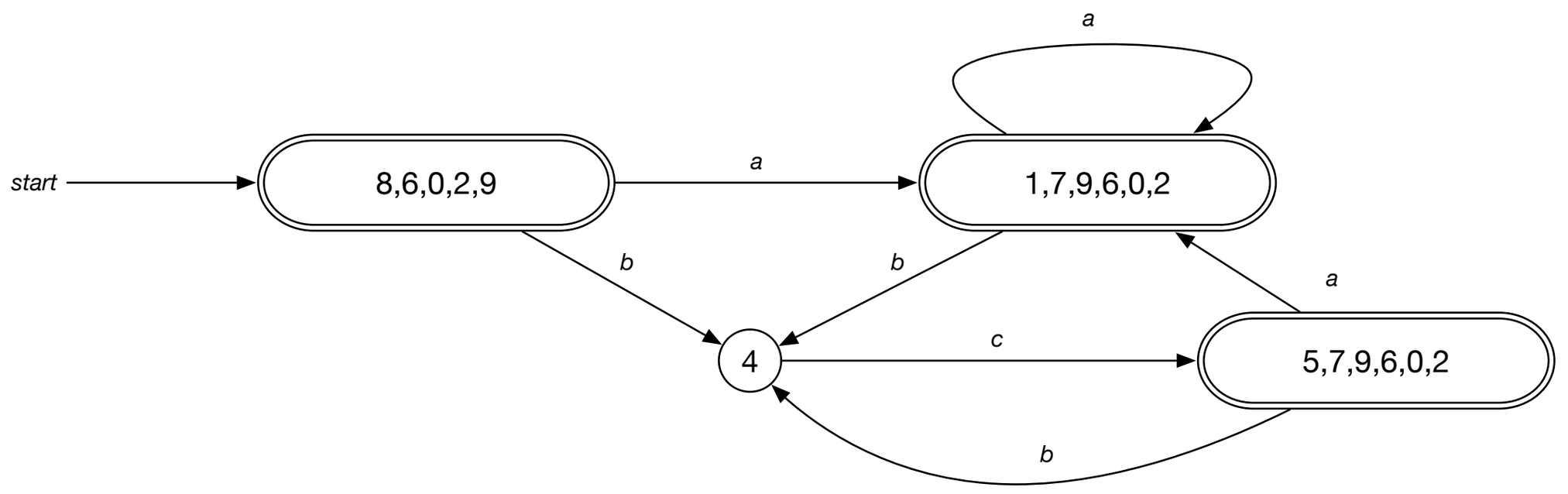


From regexp to NFA (a|bc)*

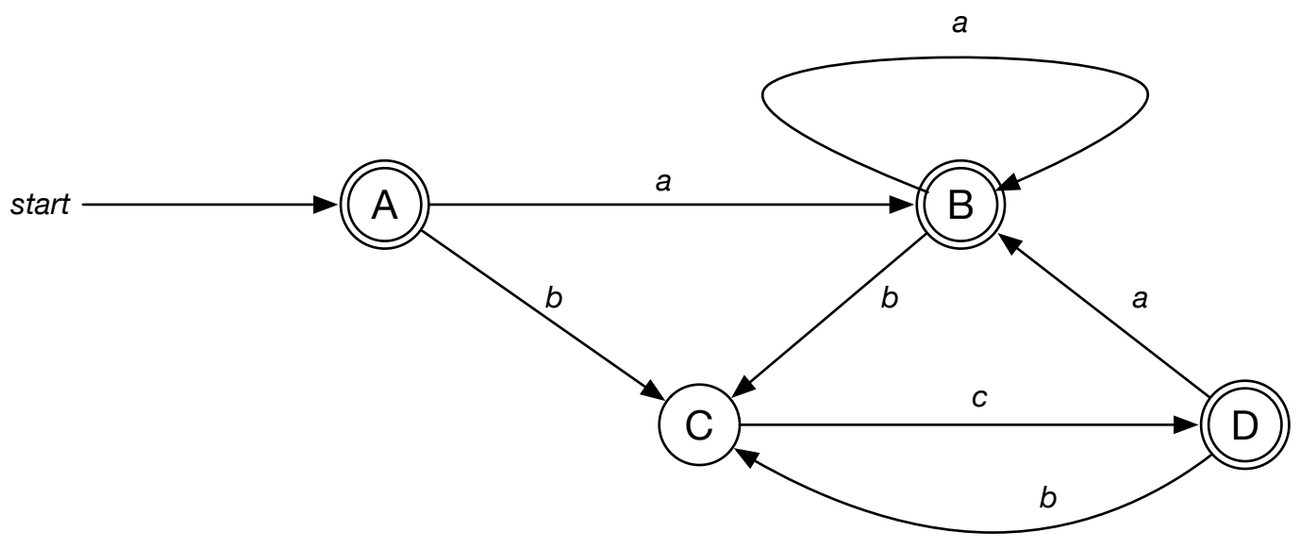


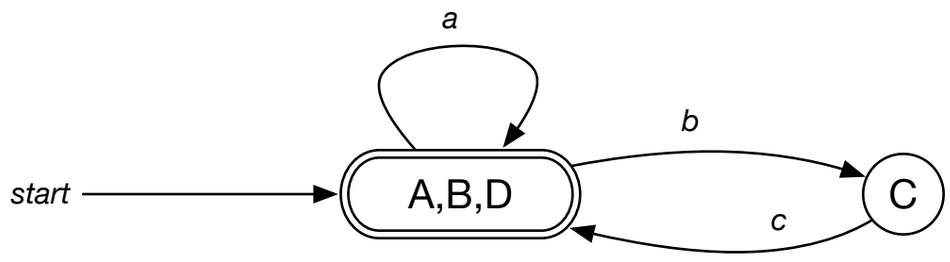
NFA to DFA / subset construction





DFA minimization

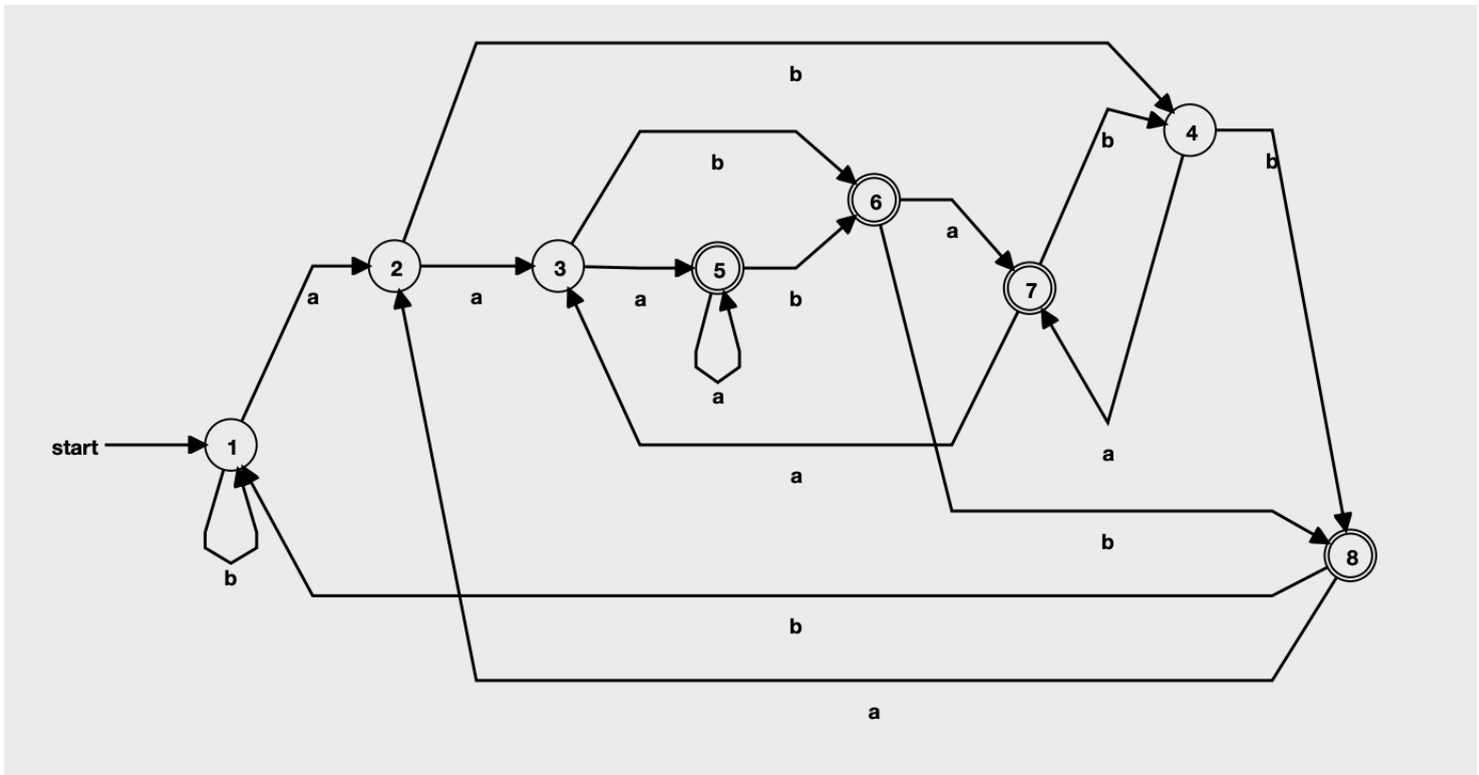




Question

The number of states in the min-DFA for regex $(a \mid b) * a(a \mid b)^n$ is exponential in n. Why?

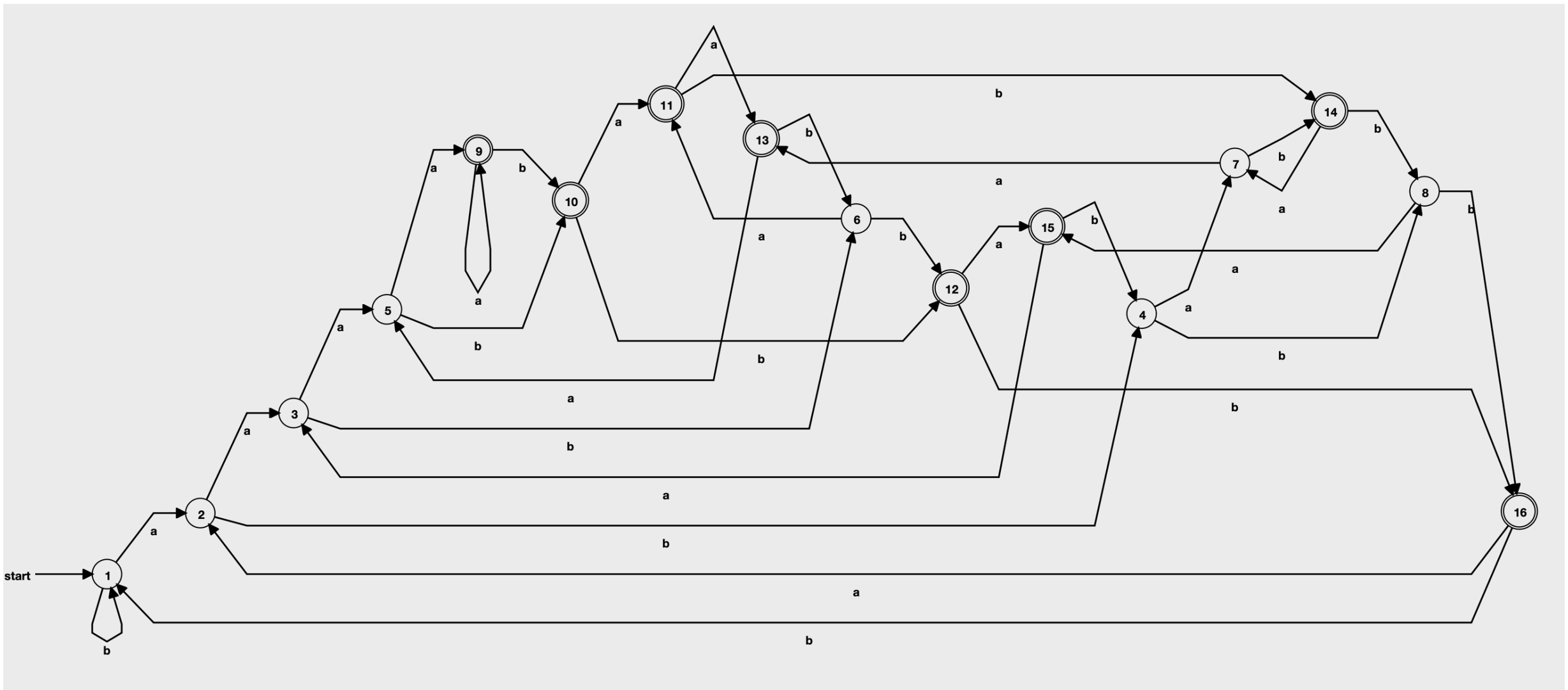
- n = 2 $(a \mid b) * a(a \mid b)(a \mid b)$



Question

The number of states in the min-DFA for regex $(a \mid b) * a(a \mid b)^n$ is exponential in n. Why?

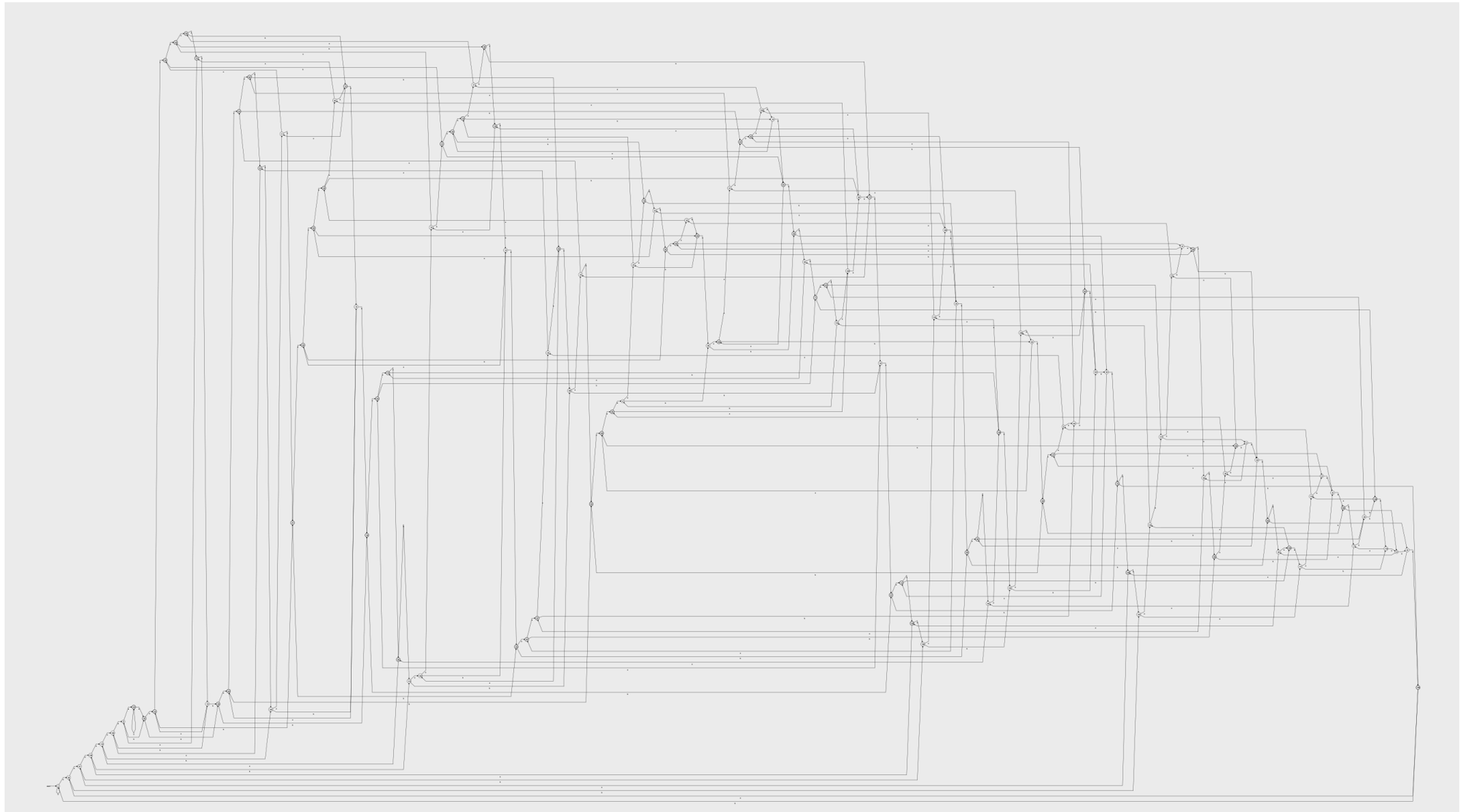
- n = 3
- $(a \mid b) * a(a \mid b)(a \mid b)(a \mid b)$



Question

The number of states in the min-DFA for regex $(a|b)*a(a|b)^n$ is exponential in n. Why?

- n = 6
- $(a|b)*a(a|b)^6$



Answer

- $(a \mid b) * a(a \mid b)^n$ accepts any string where the character at position $n + 1$ **from the end** is an a

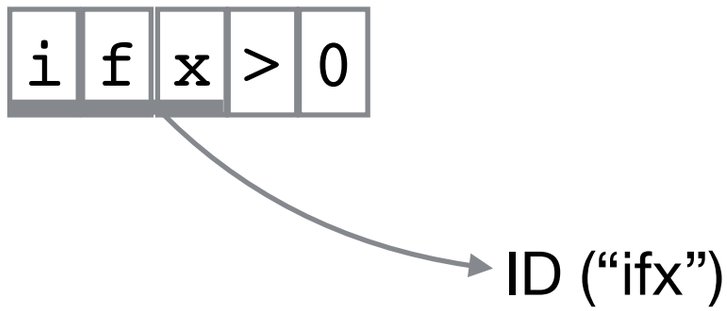
- We need to remember (in states) the $n + 1$ last character of the string as we process it

$$abba \cdots bbbababb \overbrace{\boxed{a}aaababa \cdots baaababa}^{(n+1) \quad \text{characters}}$$

- There are 2^{n+1} strings of a's and b's of length $n + 1$

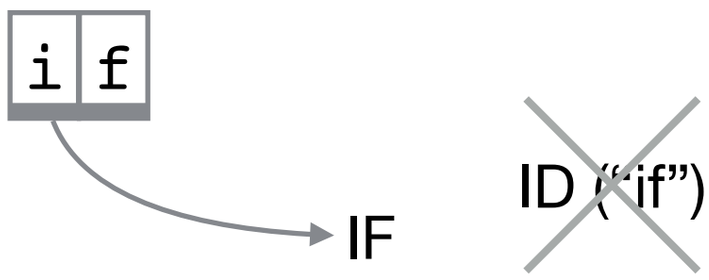
Resolving ambiguities

- Rule: when a string can match multiple tokens, the *longest* matching token wins



- `if` `{IF};`
- `[a-z][a-z0-9]*` `{ID};`

- We also need to specify *priorities* if we match several tokens of the same length.
  - Usual rule: *earliest declaration* wins



Lexical analysis

Specification: Tokens as regular exps +longest-matching rule +priorities

Formalism: NFA → DFA

Implementation: Simulate DFA

linear complexity

Output: Program that translates raw text into stream of tokens

“classical” approach – from RegEx to NFA to DFA

Lexical analysis

Specification: Tokens as regular exps +longest-matching rule +priorities

Formalism: NFA → DFA

Implementation: Simulate NFA → Simulate DFA *linear complexity*

Output: Program that translates raw text into stream of tokens

“classical” approach – from RegEx to NFA to DFA

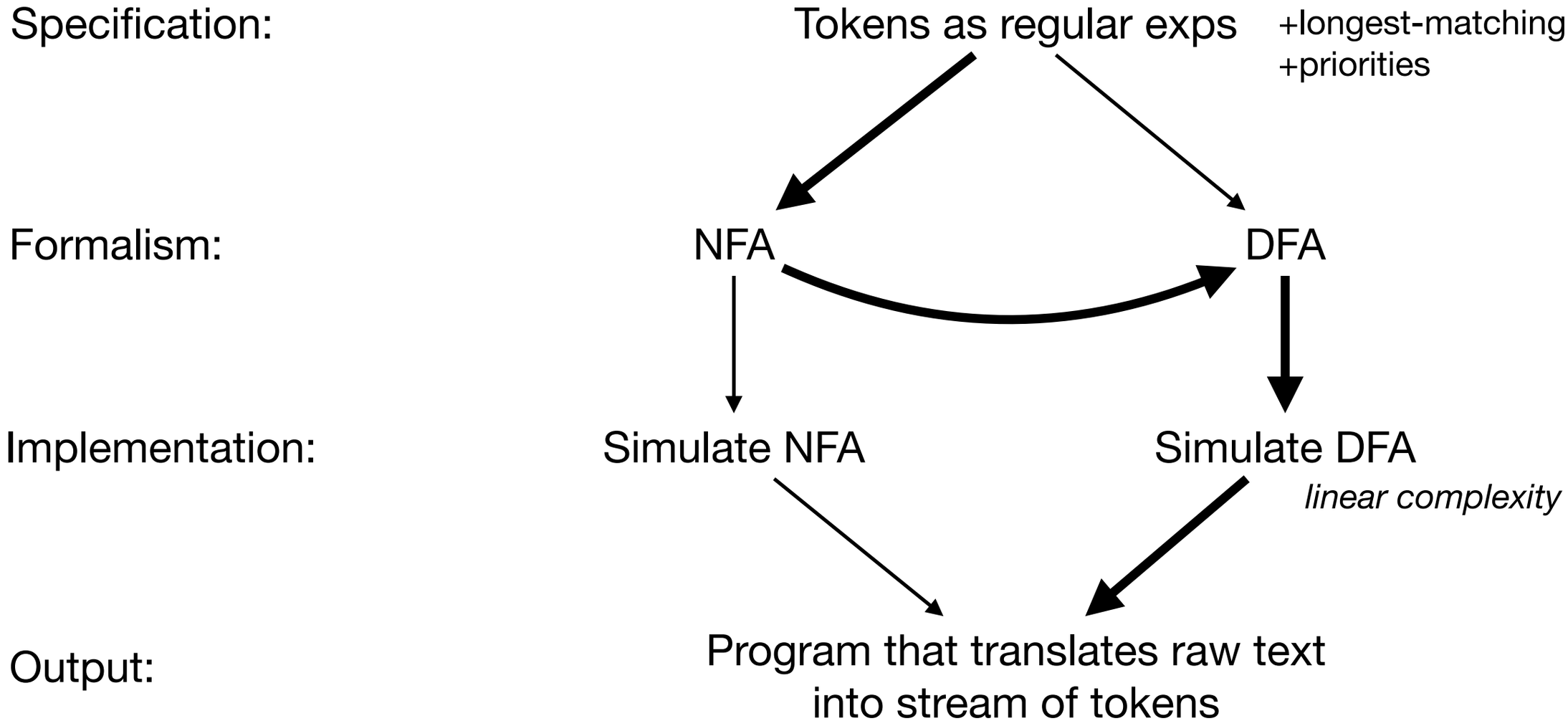
Lexical analysis

Specification: Tokens as regular exps +longest-matching rule +priorities

Formalism: NFA DFA

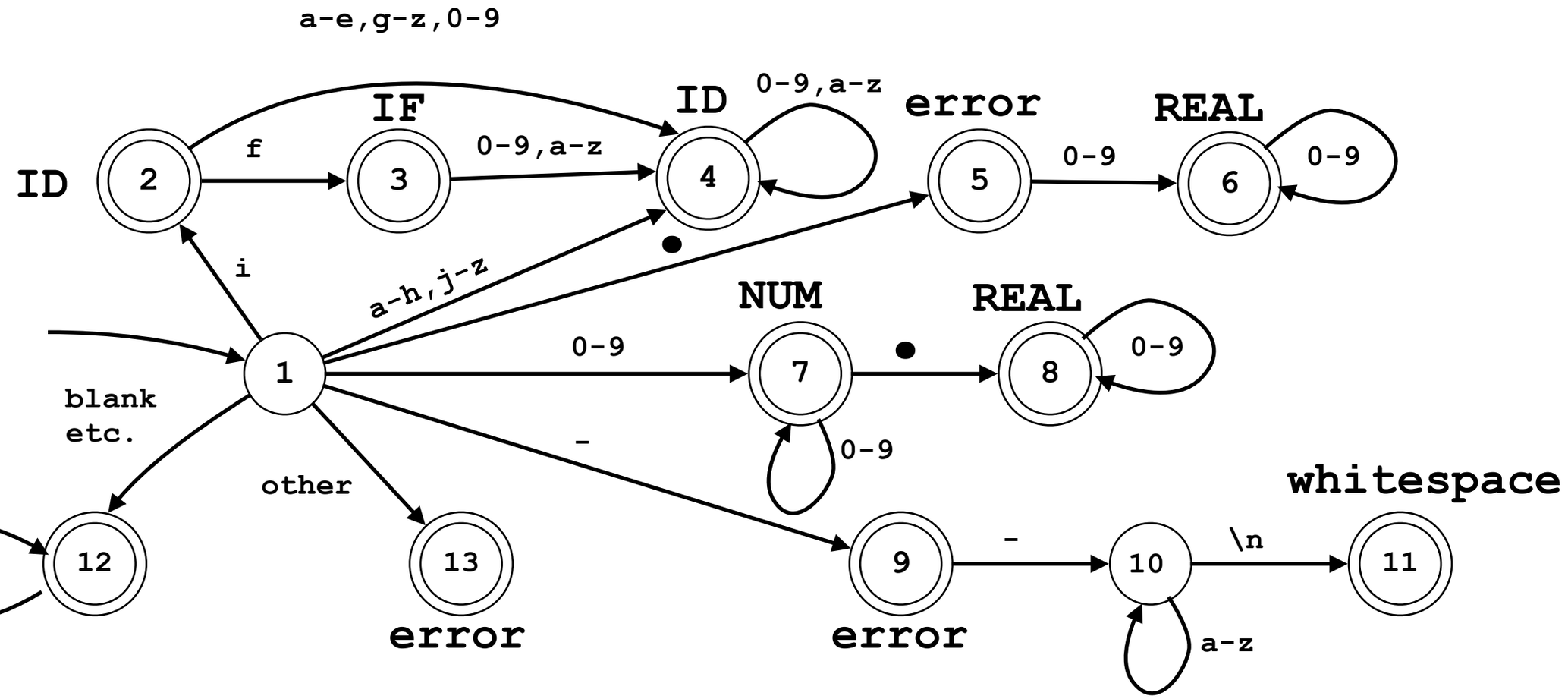
Implementation: Simulate NFA Simulate DFA *linear complexity*

Output: Program that translates raw text into stream of tokens



“classical” approach – from RegEx to NFA to DFA

Total NFA for ID,IF,NUM,REAL




```
if                                                    {IF};
[a-z][a-z0-9]*                                        {ID};
[0-9]+                                               {NUM};
([0-9]+"."[0-9]*) | ([0-9]* "." [0-9]+)             {REAL};
("--" [a-z]*"\n") | (" "|"\t")                 { continue()};
_                                          { error();continue()};
```

OCamllex

- Lexer generator, provided with OCaml
- Accepts lexical specification, produces a scanner
- Example specification

```
{
  open Parser
  exception Error of string
}

rule token = parse
  [' ' '\t' ]      { token lexbuf }   (* skip blanks by tail-calling *)
| ['0'-'9']+ as i  { INT (int_of_string i) } (* produce some token *)
| "/*"             { comment 0 lexbuf }      (* call into another entrypoint *)
| _
    { raise (Error (Printf.sprintf
            "At offset %d: unexpected character.\n" (Lexing.lexeme_start lexbuf))) }

and comment commentLevel = parse
| "/*"     { comment (commentLevel+1 ) lexbuf } (* recursively tail-call oneself *)
| "*/"     { (if commentLevel = 0 then token else comment (commentLevel-1) ) lexbuf }
| _        { comment commentLevel lexbuf }     (* contiune *)
```