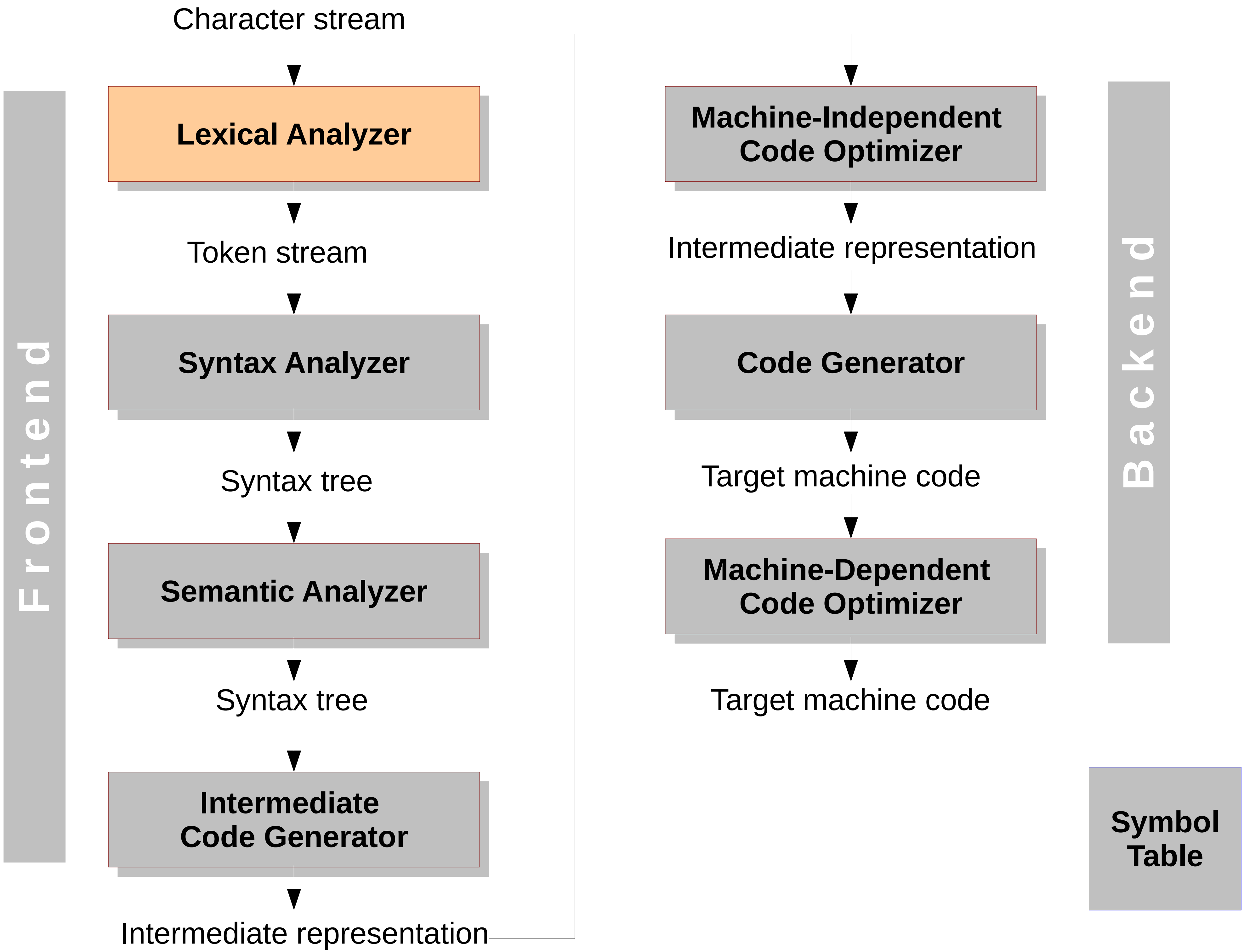
Lexing

Rupesh Nasre.

CS3300 Compiler Design
IIT Madras
Jul 2018


Character stream
Lexical Analyzer
Token stream
Syntax Analyzer
Syntax tree
Semantic Analyzer
Syntax tree
Intermediate Code Generator
Intermediate representation
Frontend
Machine-Independent Code Optimizer
Intermediate representation
Code Generator
Target machine code
Machine-Dependent Code Optimizer
Target machine code
Backend
Symbol Table

Role

- Read input characters
- Group into words (lexemes)
- Return sequence of tokens
- Sometimes
  - Eat-up whitespace
  - Remove comments
  - Maintain line number information

Token, Pattern, Lexeme

Token	Pattern	Sample lexeme
if	Characters i, f	if
comparison	<= or >= or < or > or == or !=	<=, !=
identifier	letter (letter + digit)*	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but “, surrounded by “”	“core dumped”

The following classes cover most or all of the tokens

- One token for each keyword
- Tokens for the operators, individually or in classes
- Token for identifiers
- One or more tokens for constants
- One token each for punctuation symbols

Representing Patterns

- Keywords can be directly represented (break, int).
- And so do punctuation symbols ({, +).
- Others are finite, but too many!
  - Numbers
  - Identifiers
  - They are better represented using a regular expression.
  - [a-z][a-z0-9]*, [0-9]+

Classwork: Regex Recap

- If L is a set of letters (A-Z, a-z) and D is a set of digits (0-9),
  - Find the size of the language LD.
  - Find the size of the language L U D.
  - Find the size of the language L^4.
- Write regex for real numbers
  - Without eE, without +- in exponent
  - Without eE, with +- in exponent
  - With eE, with -+ in exponent (1.89E-4)

Classwork

- Write regex for strings over alphabet {*a*, *b*} that start and end with *a*.
- Strings with third last letter as *a*.
- Strings with exactly three *b*s.
- Strings with even length.
- Homework
  - Exercises 3.3.6 from ALSU.

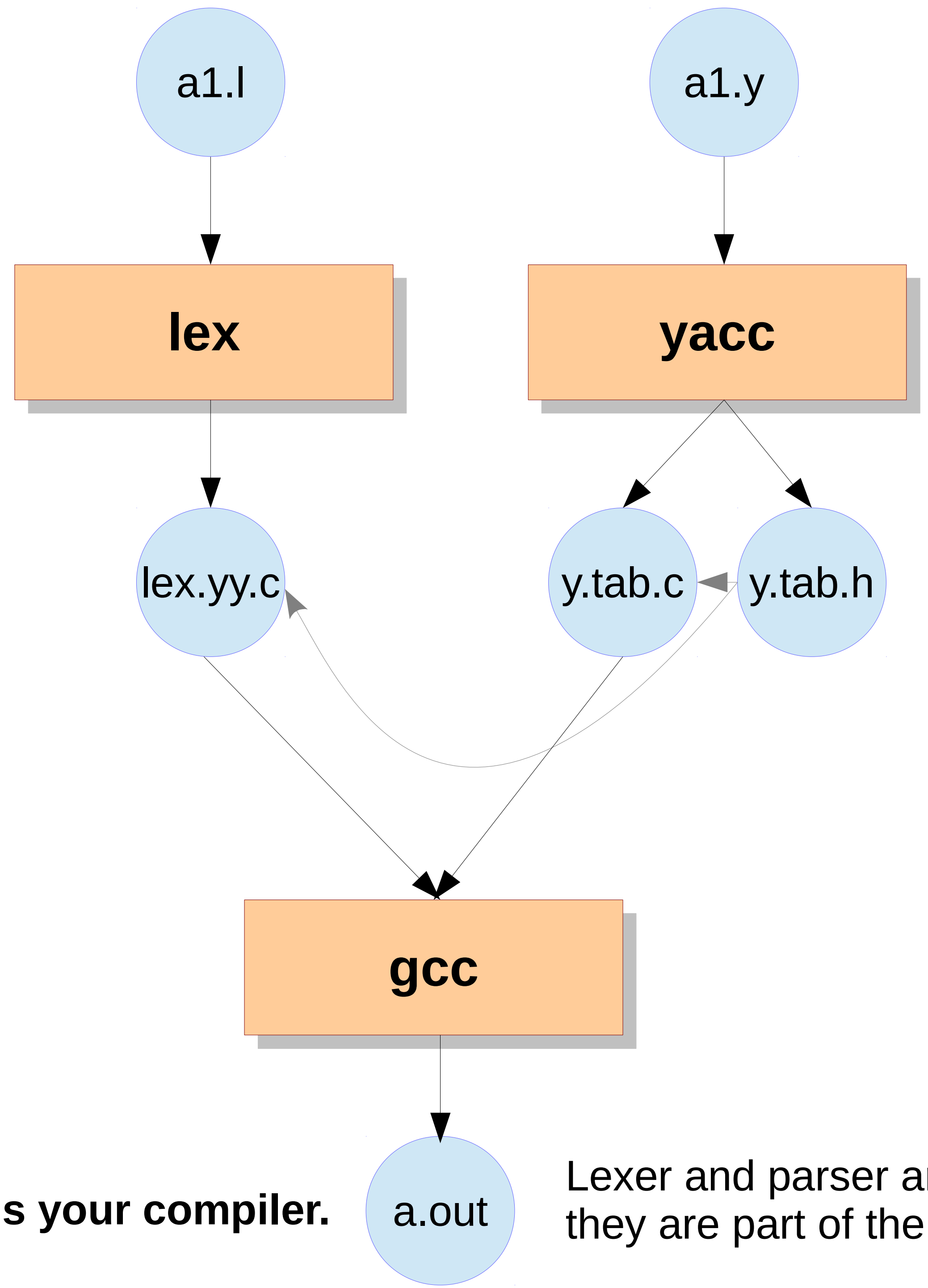
Example Lex

Patterns

```
    /* variables */
[a-z]      {
             yylval = *yytext - 'a';
             return VARIABLE;
           }

    /* integers */
[0-9]+     {
             yylval = atoi(yytext);
             return INTEGER;
           }

    /* operators */
[-+()=/*\n] { return *yytext; }

    /* skip whitespace */
[ \t]      ;

    /* anything else is an error */
.            yyerror("invalid character");
```

Tokens

Lexemes



This is your compiler.

Lexer and parser are not separate binaries; they are part of the same executable.

Lex Regex

Expression	Matches	Example
c	Character c	a
\c	Character c literally	*
“s”	String s literally	“**”
.	Any character but newline	a.*b
^	Beginning of a line	^abc
$	End of a line	abc$
[s]	Any of the characters in string s	[abc]
[^s]	Any one character not in string s	[^abc]
r*	Zero or more strings matching r	a*
r+	One or more strings matching r	a+
r?	Zero or one r	a?
r{m, n}	Between m and n occurrences of r	a{1,5}
r1r2	An r1 followed by an r2	ab
r1 | r2	An r1 or an r2	a | b
(r)	Same as r	(a | b)
r1/r2	r1 when followed by r2	abc/123

Homework

- Write a lexer to identify special words in a text.
  - Words like *stewardesses*: only one hand
  - Words like *typewriter*: only one keyboard row
  - Words like *skepticisms*: alternate hands
- Implement grep using lex with search pattern as alphabetical text (no operators *, ?, ., etc.).

Lexing and Context

- Language design should ensure that lexing can be done without context.
- Your assignments and most languages need context-insensitive lexing.

```
DO 5  I = 1.25
```

```
DO 5  I = 1,25
```

- “DO 5 I” is an identifier in Fortran, as spaces are allowed in identifiers.
- Thus, first is an assignment, while second is a loop.
- Lexer doesn't know whether to consider the input “DO 5 I” as an identifier or as a part of the loop, until parser informs it based on dot or comma.
- Alternatively, lexer may employ a lookahead.

Lexical Errors

- It is often difficult to report errors for a lexer.
  - fi (a == f(x)) ...
  - A lexer doesn't know the context of fi. Hence it cannot “see” the structure of the sentence – structure is known only to the parser.
  - fi = 2; OR fi(a == f(x));
- But some errors a lexer can catch.
  - 23 = @a;
  - if $x friendof anil ...

What should a lexer do on catching an error?

Error Handling

- Multiple options
  - exit(1);
  - Panic mode recovery: delete enough input to recognize a token
  - Delete one character from the input
  - Insert a missing character into the remaining input
  - Replace a character by another character
  - Transpose two adjacent characters
- In practice, most lexical errors involve a single character.
- Theoretical problem: Find the smallest number of transformations (add, replace, delete) needed to convert the source program into one that consists only of valid lexemes.
  - Too expensive in practice to be worth the effort.

Homework

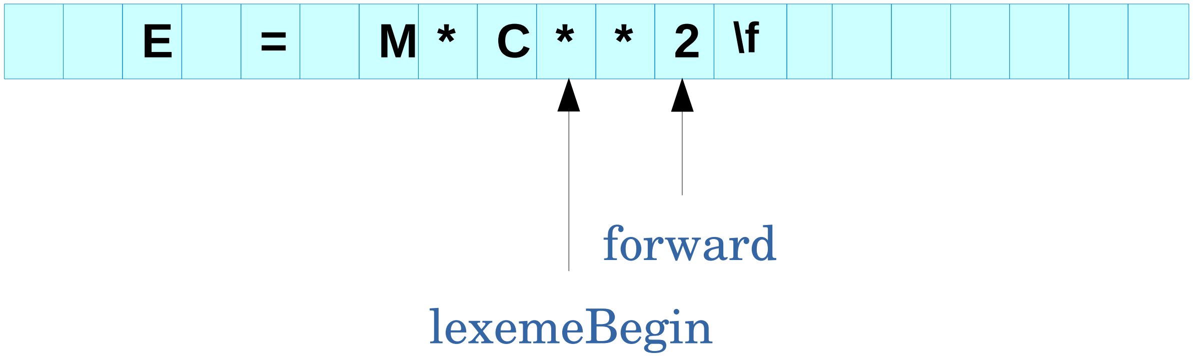
- Try exercise 3.1.2 from ALSU.

Input Buffering

- “*We cannot know we were executing a finite loop until we come out of the loop.*”
- In C, without reading the next character we cannot determine a binary minus symbol (a-b).
  - ->, -=, --, -e, ...
  - Sometimes we may have to look several characters in future, called *lookahead*.
  - In the fortran example (DO 5 I), the lookahead could be upto dot or comma.
- Reading character-by-character from disk is inefficient. Hence buffering is required.

Input Buffering

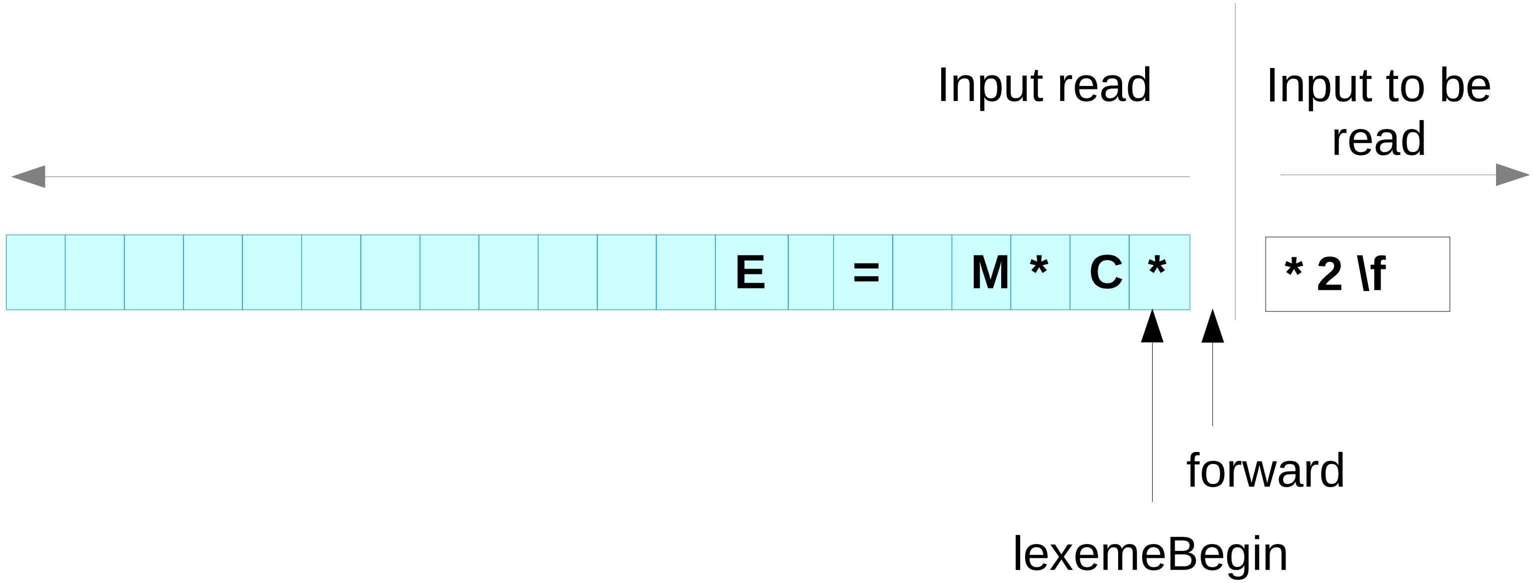
- A block of characters is read from disk into a buffer.
- Lexer maintains two pointers:
  - lexemeBegin
  - forward



What is the problem with such a scheme?

Input Buffering

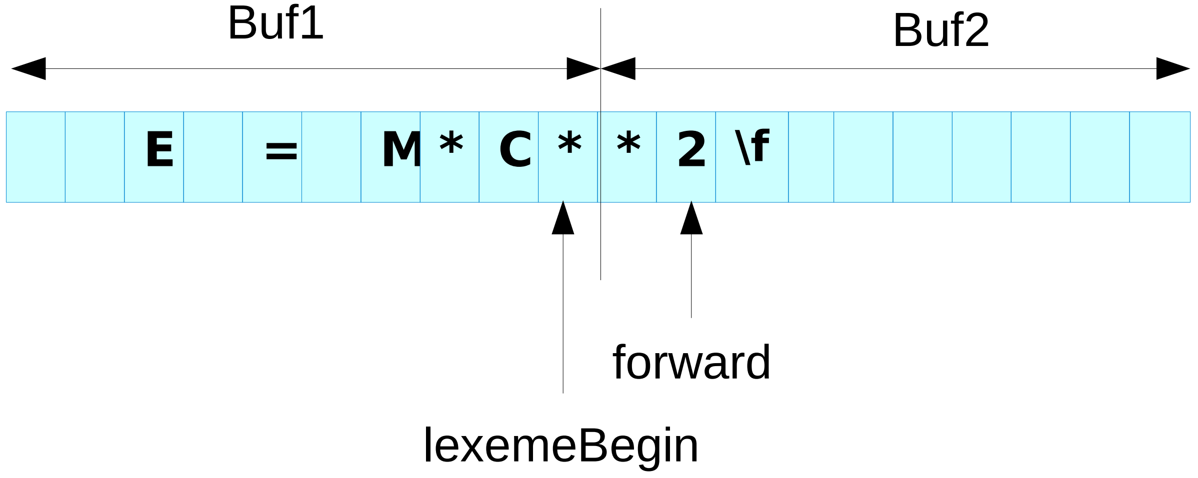
- The issue arises when the lookahead is beyond the buffer.
- When you load the buffer, the previous content is overwritten!



How do we solve this problem?

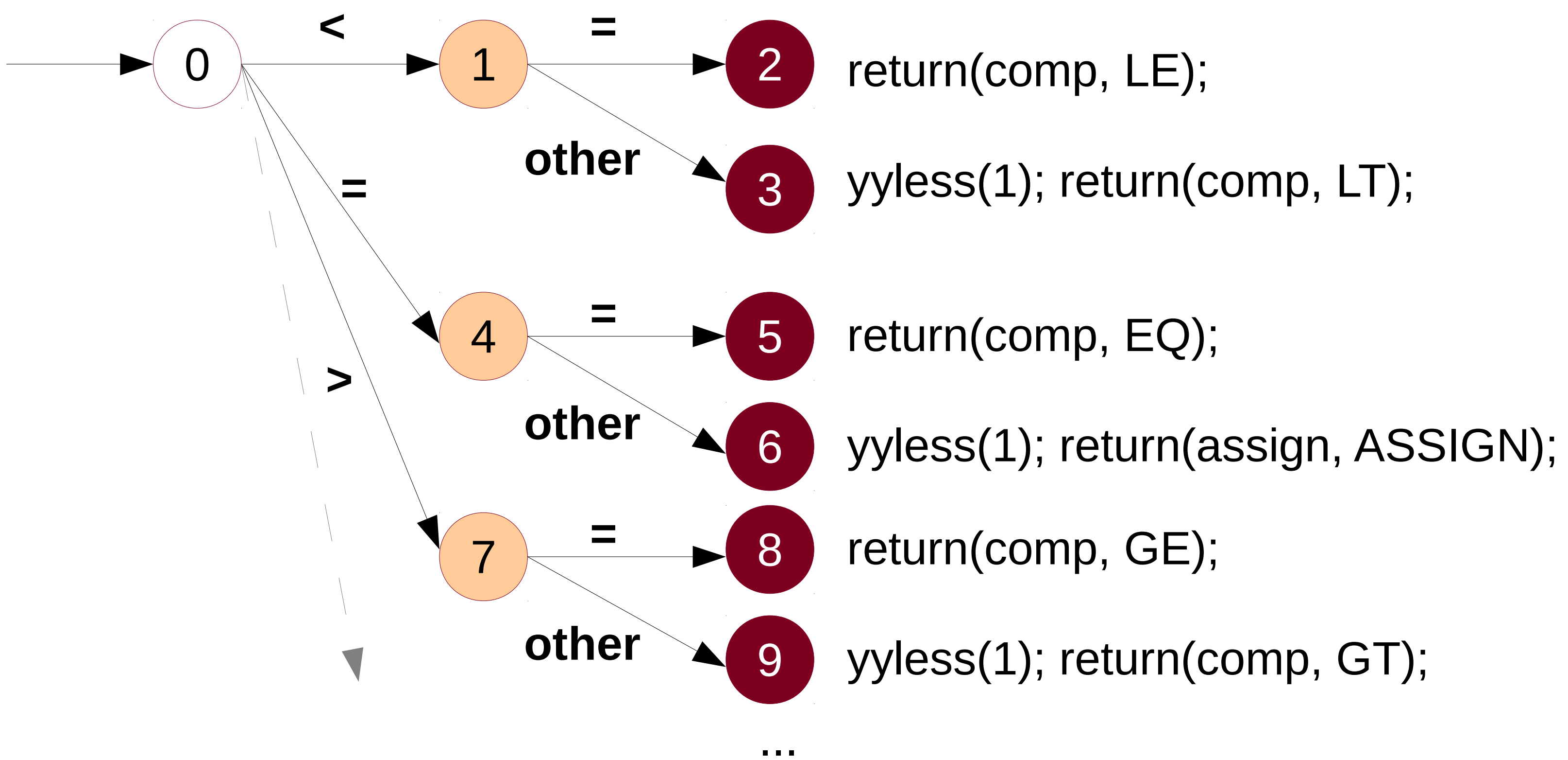
Double Buffering

- Uses two (half) buffers.
- Assumes that the lookahead would not be more than one buffer size.



Transition Diagrams

- Step to be taken on each character can be specified as a state transition diagram.
  - Sometimes, action may be associated with a state.



Keywords vs. Identifiers

- Keywords may match identifier pattern
  - Keywords: int, const, break, ...
  - Identifiers: (alpha | _) (alpha | num | _)*
- If unaddressed, may lead to strange errors.
  - Install keywords a priori in the symbol table.
  - Prioritize keywords
- In lex, the rule for a keyword must precede that of the identifier.

```
[a-z_A-Z][a-zA-Z_0-9]*      { return IDENT; }

"break"                     { return BREAK; }
```

Incorrect (lex may give warning)

```
"break"                     { return BREAK; }

[a-z_A-Z][a-zA-Z_0-9]*      { return IDENT; }
```

Correct

Special vs. General

- In general, a specialized pattern must precede the general pattern (*associativity*).
- Lex also follows maximum substring matching rule (*precedence*).
  - Reordering the rules for < and <= would not affect the functionality.
- Compare with rule specialization in Prolog.
- **Classwork**: Count number of *he* and *she* in a text.
- **Classwork**: Write lex rules to recognize quoted strings in C.
  - Try to recognize \” inside it.

he and she

```
she   ++s;
he         ++h;
```

```
she   { ++s; REJECT; }
he        { ++h; }
```

Retries another rule

What if I want to count all possible substrings *he*?

In general, the action associated with a rule may not be easy / modular to duplicate.

Input: he ahe he she she fsfds fsf fs sfhe he she she she

he=5, she=5

he=10, she=5

By the way...

- Sometimes, you need not have a parser at all...
  - You could define *main* in your lex file.
  - Simply call *yylex()* from *main*.
  - Compile using *lex*, then compile *lex.yy.c* using *gcc* and execute *a.out*.

Lookahead

Duniya usi ki hai jo aage dekhe

Lookahead

- Lexer needs to look into the future to know where it is presently.

DO 5 I = 1,25

DO / .* COMMA { return DO;}

- / signifies the lookahead symbol. The input is read and matched, but is left unconsumed in the current rule.

Corollary: DO loop index and increment must be on the same line – no arbitrary whitespace allowed.

String Matching

- Lexical analyzer relies heavily on string matching.
- Given a program text T (length n) and a pattern string s (length m), we want to check if s occurs in T.
- A naive algorithm would try all positions of T to check for s (complexity m*n).

Can we do better?

Where can we do better?

- T = abababaababbbabbababb
- s = ababaa

i = 0

abababaababbbabbababb

ababaa

Where can we do better?

- T = ababababaababbbabbbababb
- s = ababaa

i = 0

ababababaababbbabbbababb

ababaa

Where can we do better?

- T = ababaabababbbabbababb
- s = ababaa

i = 1

ababaabababbbabbababb

ababaa

Where can we do better?

- T = abababaababbbabbababb
- s = ababaa

i = 2

abababaababbbabbababb

ababaa

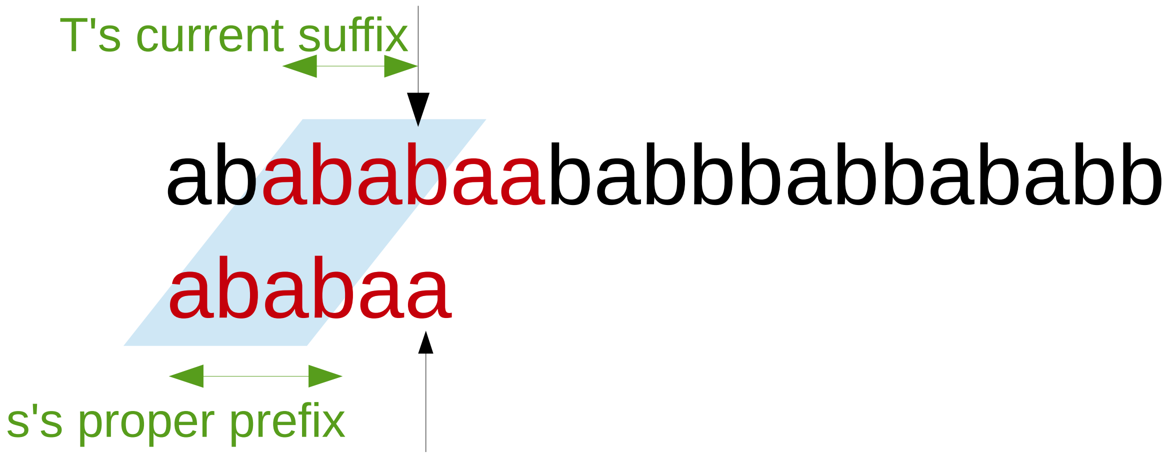
Match found

We need to handle the failure better.

Where can we do better?

- T = abababaababbbabbababb
- s = ababaa

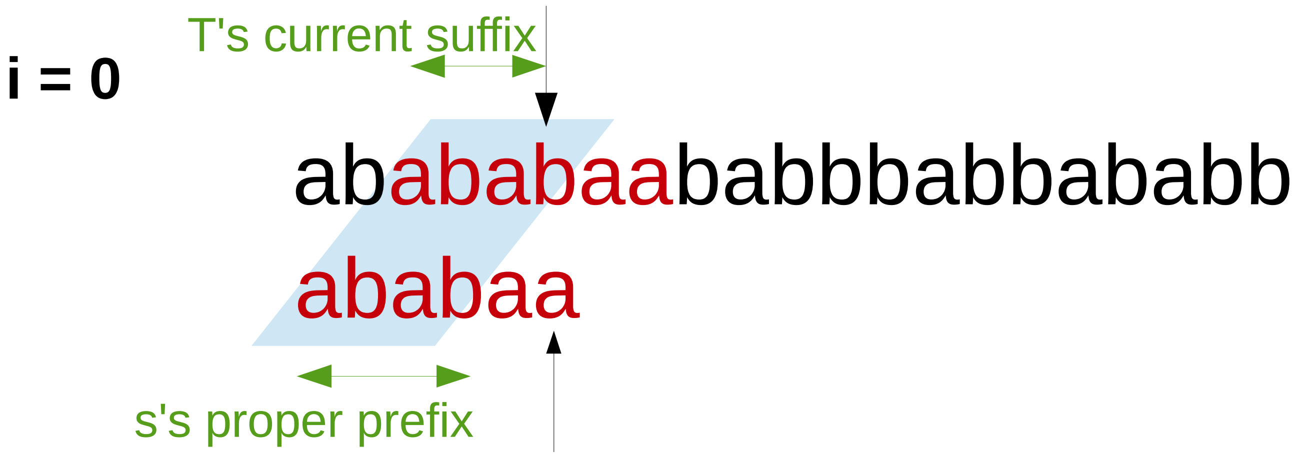
i = 0



Key observation: T's current suffix which is a proper prefix in s has the treasure for us.
Whenever there is a mismatch, we should utilize this overlap, rather than restarting.

Where can we do better?

- T = abababaababbbabbababb
- s = ababaa



Key observation: T's current suffix which is a proper prefix in s has the treasure for us.
Whenever there is a mismatch, we should utilize this overlap, rather than restarting.

KMP

- Knuth-Morris-Pratt algorithm for string matching.
- Whenver there is a mismatch, do not restart; rather *fail intelligently*.
- We define a failure function for each position, taking into account the suffix and the prefix.
- Note that the matched part of the large string T is essentially the pattern string s. Thus, failure function can be computed simply using pattern s.

Failure is not final.

Failure function for *ababaa*

i	1	2	3	4	5	6
f(i)	0	0	1	2	3	1

seen	a	ab	aba	abab	ababa	ababaa
prefix	ϵ	ϵ	a	ab	aba	a

Algorithm given as Figure 3.19 in ALSU.

String matching with failure function

Text = $a_1 a_2 \ldots a_m$; pattern = $b_1 b_2 \ldots b_n$ (both indexed from 1)

```
s = 0
for (i = 1; i <= m; ++i) {          ◄── Go over Text
    if (s > 0 && a_i != b_{s+1}) s = f(s)   ◄── Handle failure
    if (a_i == b_{s+1}) ++s          ◄── Character match
    if (s == n) return "yes"         ◄── Full match
}
return "no"
```

i	1	2	3	4	5	6
f(i)	0	0	1	2	3	1

seen	a	ab	aba	abab	ababa	ababaa
prefix	ϵ	ϵ	a	ab	aba	a

Find the flaw in the algorithm.

String matching with failure function

Text = $a_1a_2...a_m$; pattern = $b_1b_2...b_n$ (both indexed from 1)

s = 0
for (i = 1; i <= m; ++i) { ← Go over Text
  while (s > 0 && a_i != b_{s+1}) s = f(s) ← Handle failure
  if (a_i == b_{s+1}) ++s ← Character match
  if (s == n) return "yes" ← Full match
}
return "no"

ababab**aa**babbbabbababb

ababaa

i	1	2	3	4	**5**	6
f(i)	0	0	1	2	**3**	1

Classwork

- Find failure function for pattern *ababaa*.
- Test it on string *abababbaa*.

- Fibonacci strings are defined as
  - $s_1 = b$, $s_2 = a$, $s_k = s_{k-1}s_{k-2}$ for $k > 2$
  - e.g., $s_3 = ab$, $s_4 = aba$, $s_5 = abaab$
- Find the failure function for s_6.

Fibonacci Strings

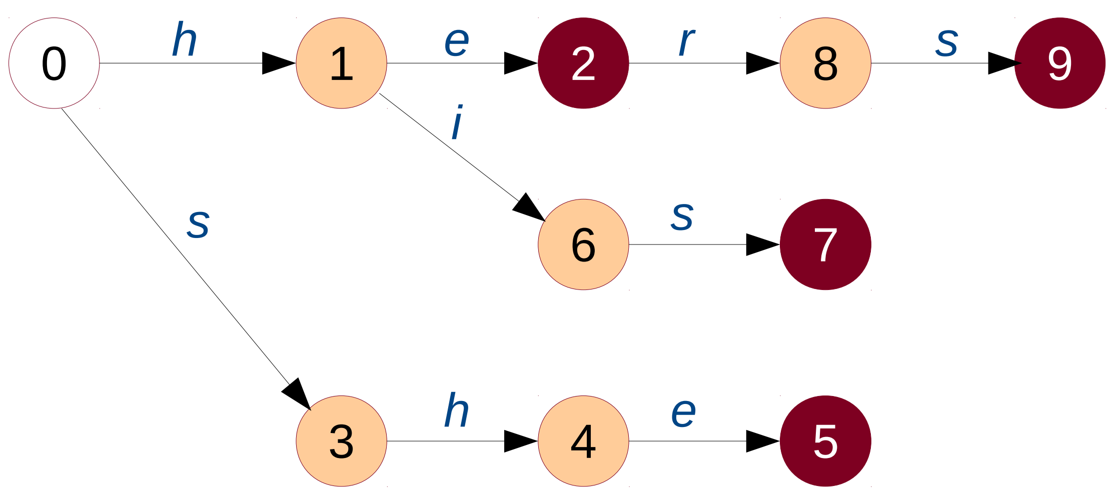
- $s_1 =$ *b*, $s_2 =$ *a*, $s_k = s_{k-1}s_{k-2}$ for $k > 2$
- e.g., $s_3 =$ *ab*, $s_4 =$ *aba*, $s_5 =$ *abaab*

- Do not contain *bb* or *aaa*.
- The words end in *ba* and *ab* alternatively.
- Suppressing last two letters creates a palindrome.
- ...

Source: Wikipedia

KMP Generalization

- KMP can be used for keyword matching.
- Aho and Corasick generalized KMP to recognize any of a set of keywords in a text.

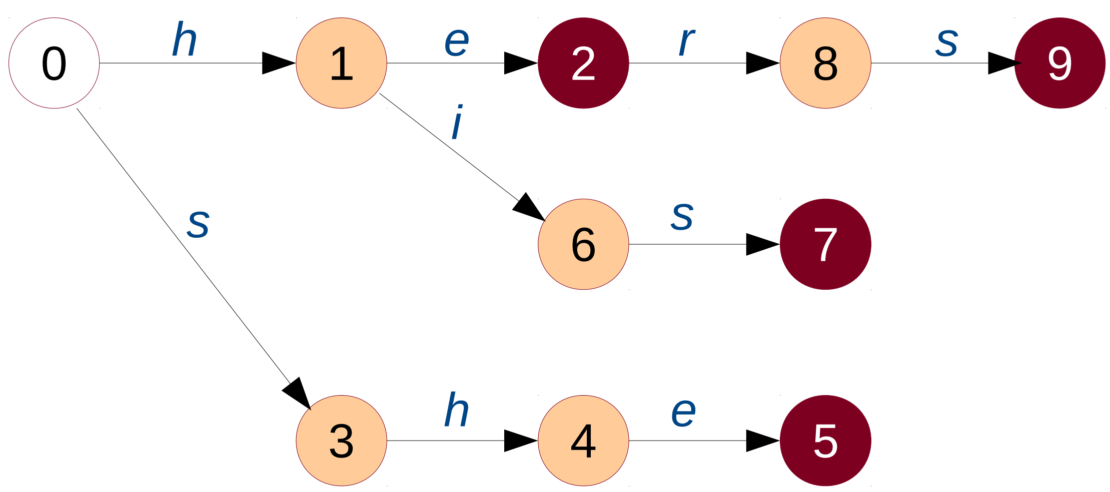


Transition diagram for keywords *he*, *she*, *his* and *hers*.

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

KMP Generalization

- When in state *i*, the failure function *f(i)* notes the state corresponding to the longest proper suffix that is also a prefix of some keyword.



Transition diagram for keywords *he*, *she*, *his* and *hers*.

i	1	2	3	4	5	6	7	8	9
f(i)	0	0	0	1	2	0	3	0	3

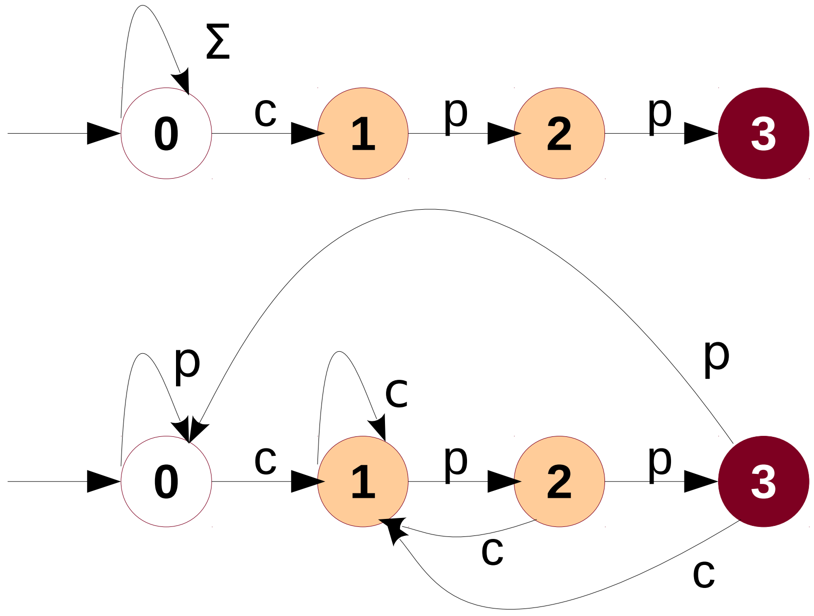
In state **7**, character s matches prefix of the keyword she to reach state **3**.

Regex to DFA

- Approach 1: Regex → NFA → DFA
- Approach 2: Regex → DFA
  - The ideas would be helpful in parsing too.

Regex → NFA → DFA

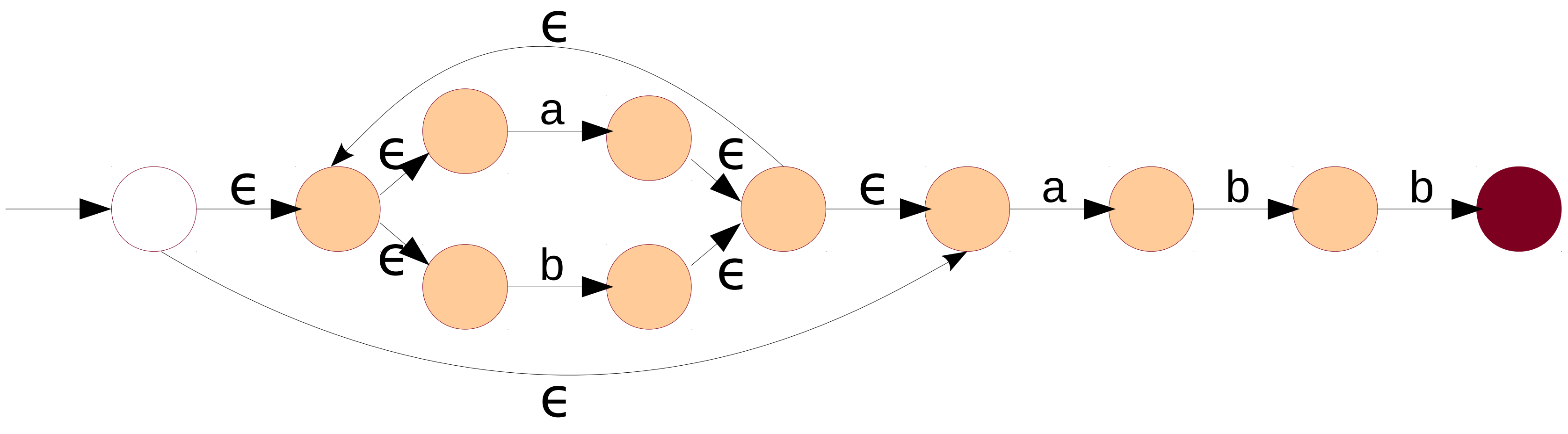
Draw an NFA for **cpp*



How does a machine draw an NFA for an arbitrary regular expression such as ((aa)*b(bb)*(aa)*)* ?

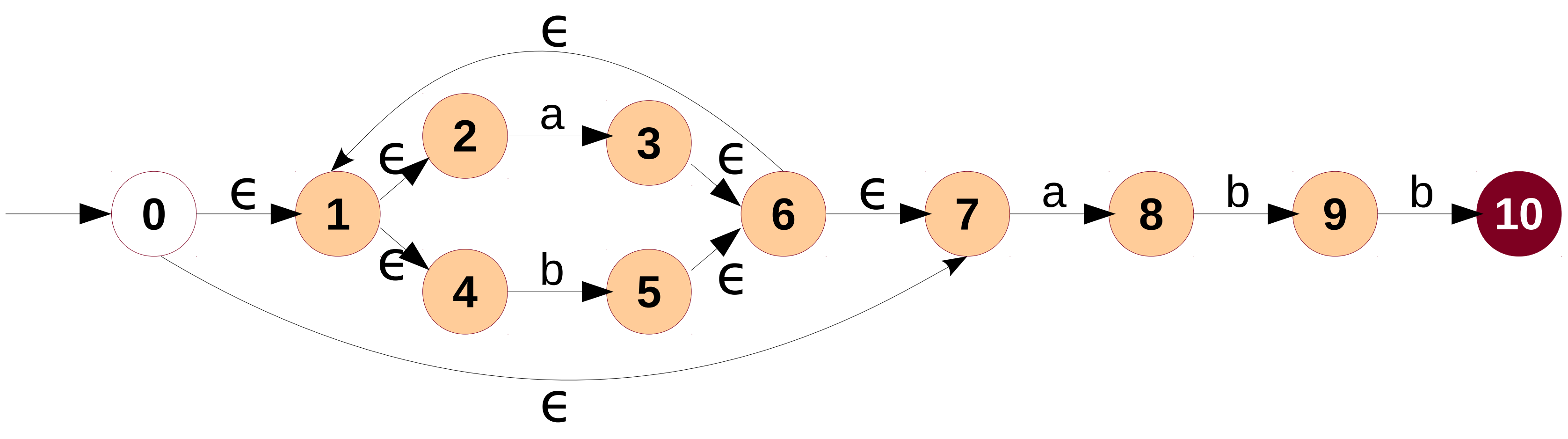
Regex → NFA → DFA

- For the sake of convenience, let's convert *cpp into *abb and restrict to alphabet {a, b}.
- Thus, the regex is (a|b)*abb.
- How do we create an NFA for (a|b)*abb?



Regex → NFA → DFA

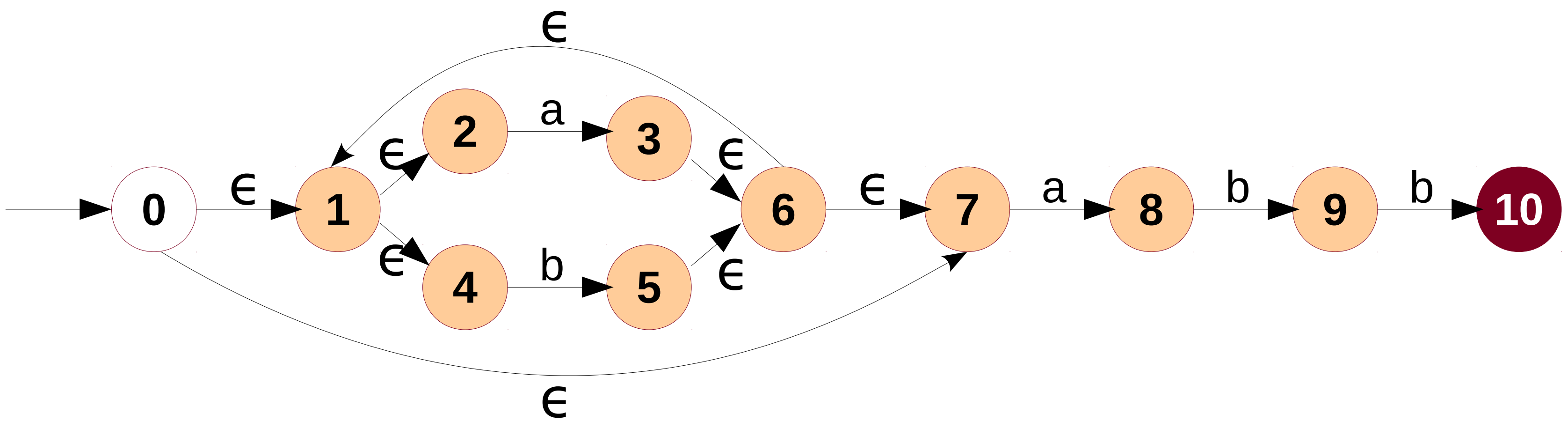
- For the sake of convenience, let's convert *cpp into *abb and restrict to alphabet {a, b}.
- Thus, the regex is (a|b)*abb.
- How do we create an NFA for (a|b)*abb?



Regex → NFA → DFA

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

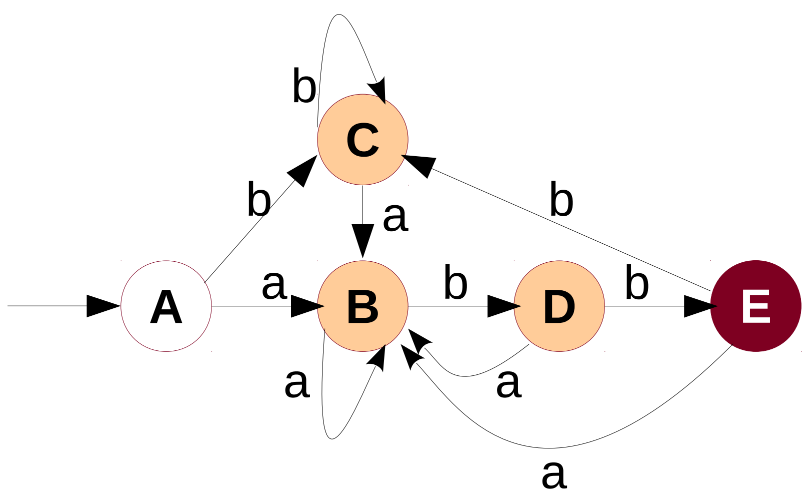
State Transition Table



Regex → NFA → DFA

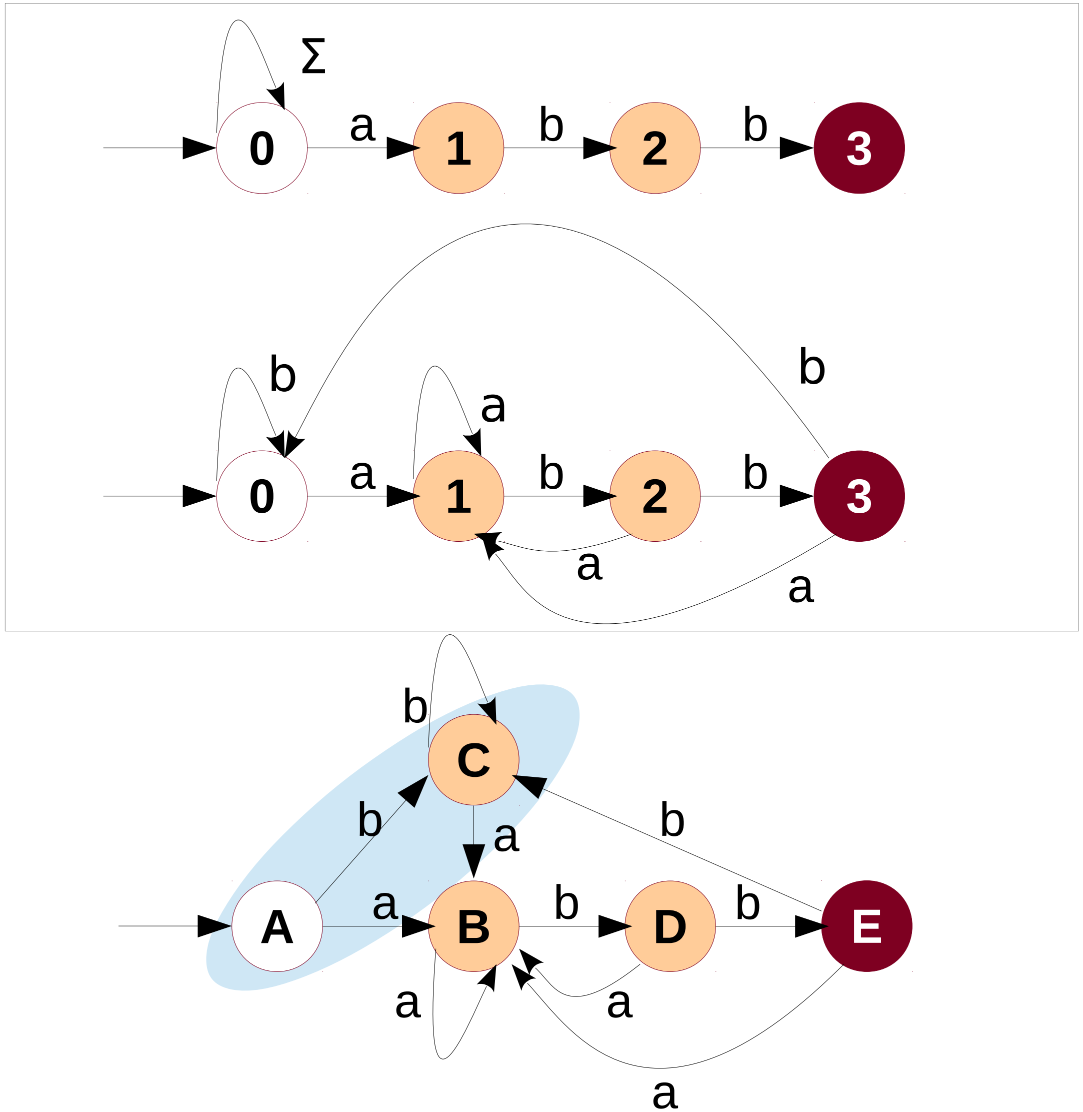
NFA state	DFA state	a	b
{0, 1, 2, 4, 7}	A	B	C
{1, 2, 3, 4, 6, 7, 8}	B	B	D
{1, 2, 4, 5, 6, 7}	C	B	C
{1, 2, 4, 5, 6, 7, 9}	D	B	E
{1, 2, 4, 5, 6, 7, 10}	E	B	C

State Transition Table



DFA

Regex → NFA → DFA



NFA

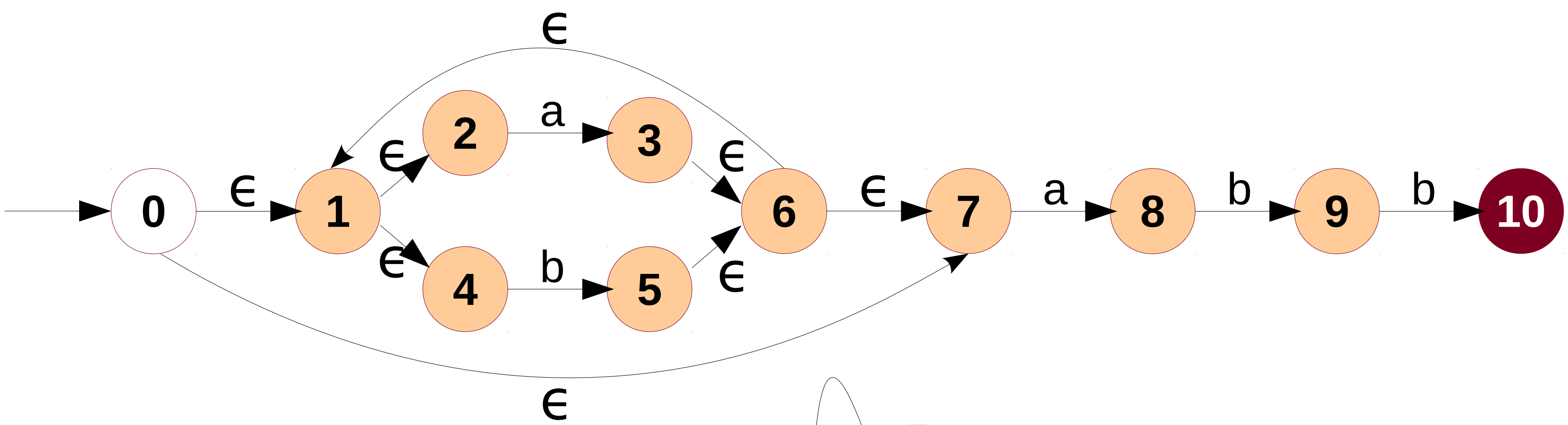
DFA

DFA
non-minimal

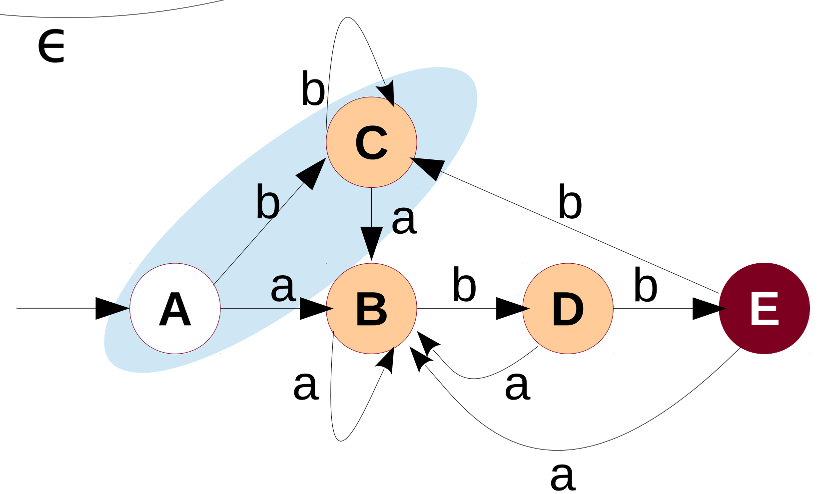
Regex → NFA → DFA

(a|b)*abb

Regex



NFA



DFA

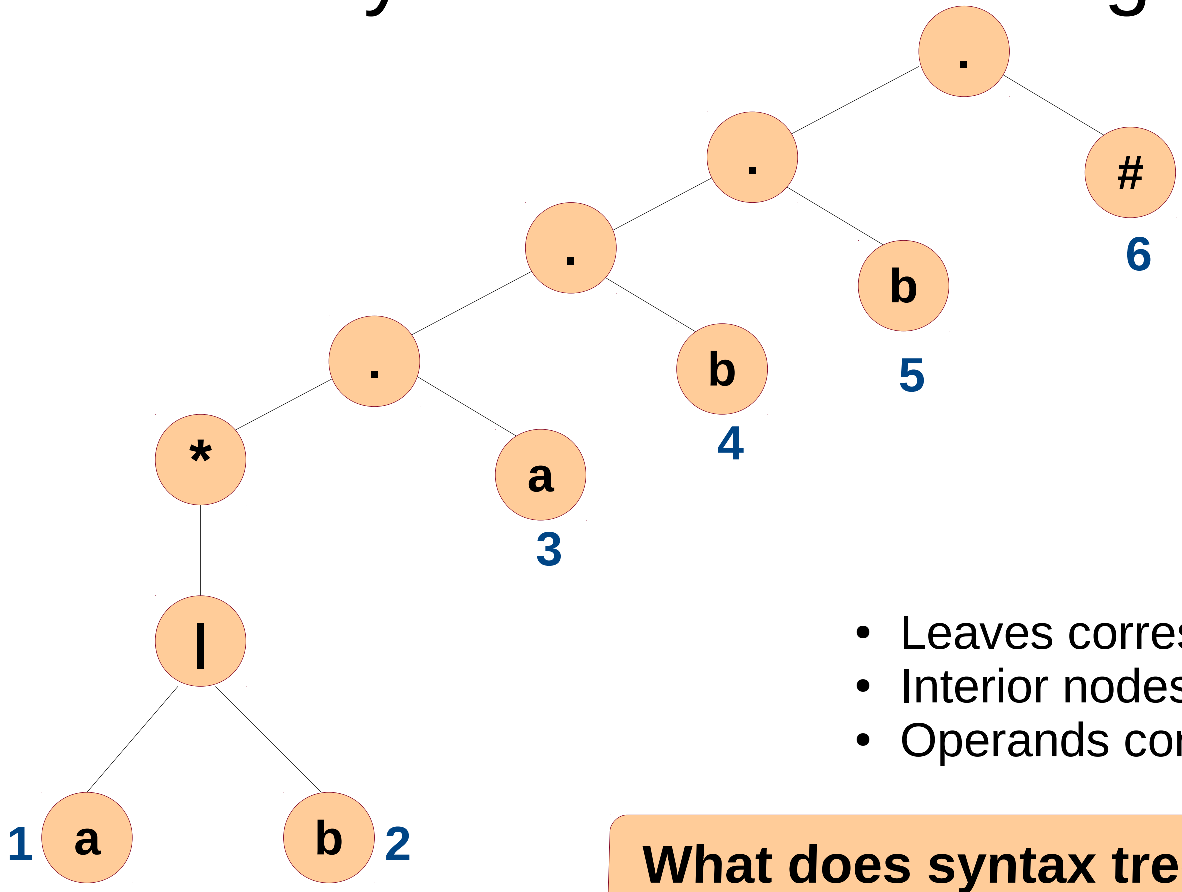
non-minimal

Regex → DFA

1. Construct a syntax tree for regex#.
2. Compute *nullable*, *firstpos*, *lastpos*, *followpos*.
3. Construct DFA using transition function.
4. Mark *firstpos(root)* as start state.
5. Mark states that contain position of # as accepting states.

Regex → DFA

- Regex is (a|b)*abb#.
- Construct a syntax tree for the regex.

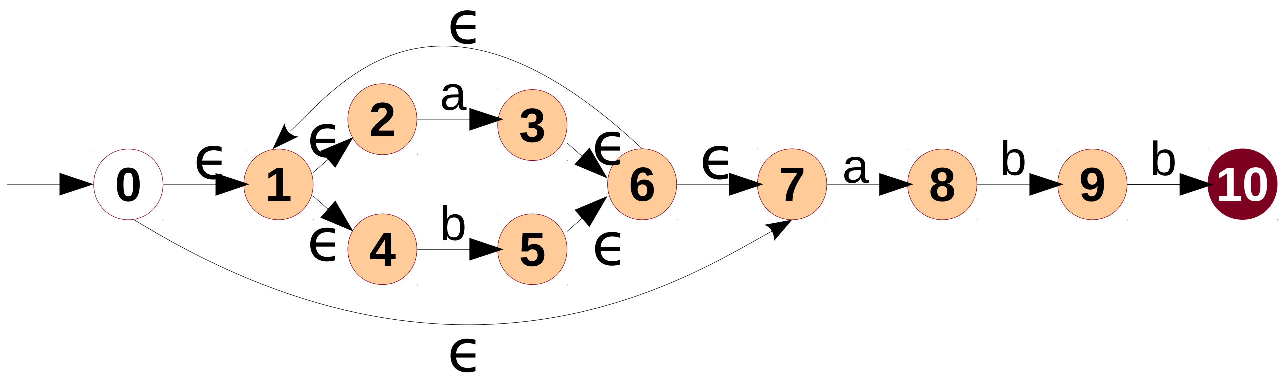


- Leaves correspond to operands.
- Interior nodes correspond to operators.
- Operands constitute strings.

What does syntax tree for regex indicate?

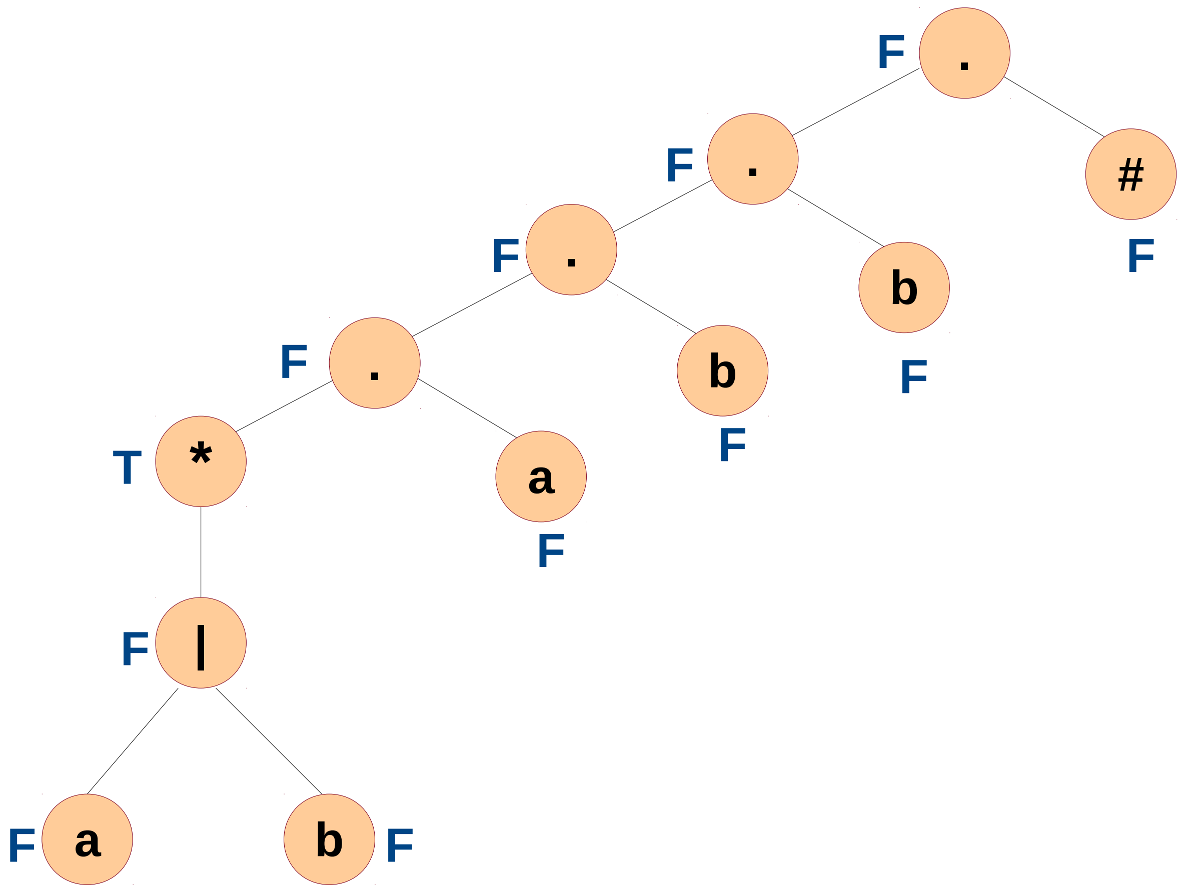
Functions from Syntax Tree

- For a syntax tree node n
  - *nullable*(n): true if n represents ϵ.
  - *firstpos*(n): set of positions that correspond to the first symbol of strings in n's subtree.
  - *lastpos*(n): set of positions that correspond to the last symbol of strings in n's subtree.
  - *followpos*(n): set of next possible positions from n for valid strings.



nullable

- Regex is (a|b)*abb#.



nullable

Node n	nullable(n)
leaf labeled ϵ	true
leaf with position i	false
or-node n = c1 | c2	nullable(c1) or nullable(c2)
cat-node n = c1c2	nullable(c1) and nullable(c2)
star-node n = c*	true

Classwork: Write down the rules for firstpos(n).

- *firstpos*(n): set of positions that correspond to the first symbol of strings in n's subtree.

firstpos

Node n	**firstpos(n)**
leaf labeled ϵ	{ }
leaf with position i	{i}
or-node n = c1 | c2	firstpos(c1) U firstpos(c2)
cat-node n = c1c2	
star-node n = c*	firstpos(c)

firstpos

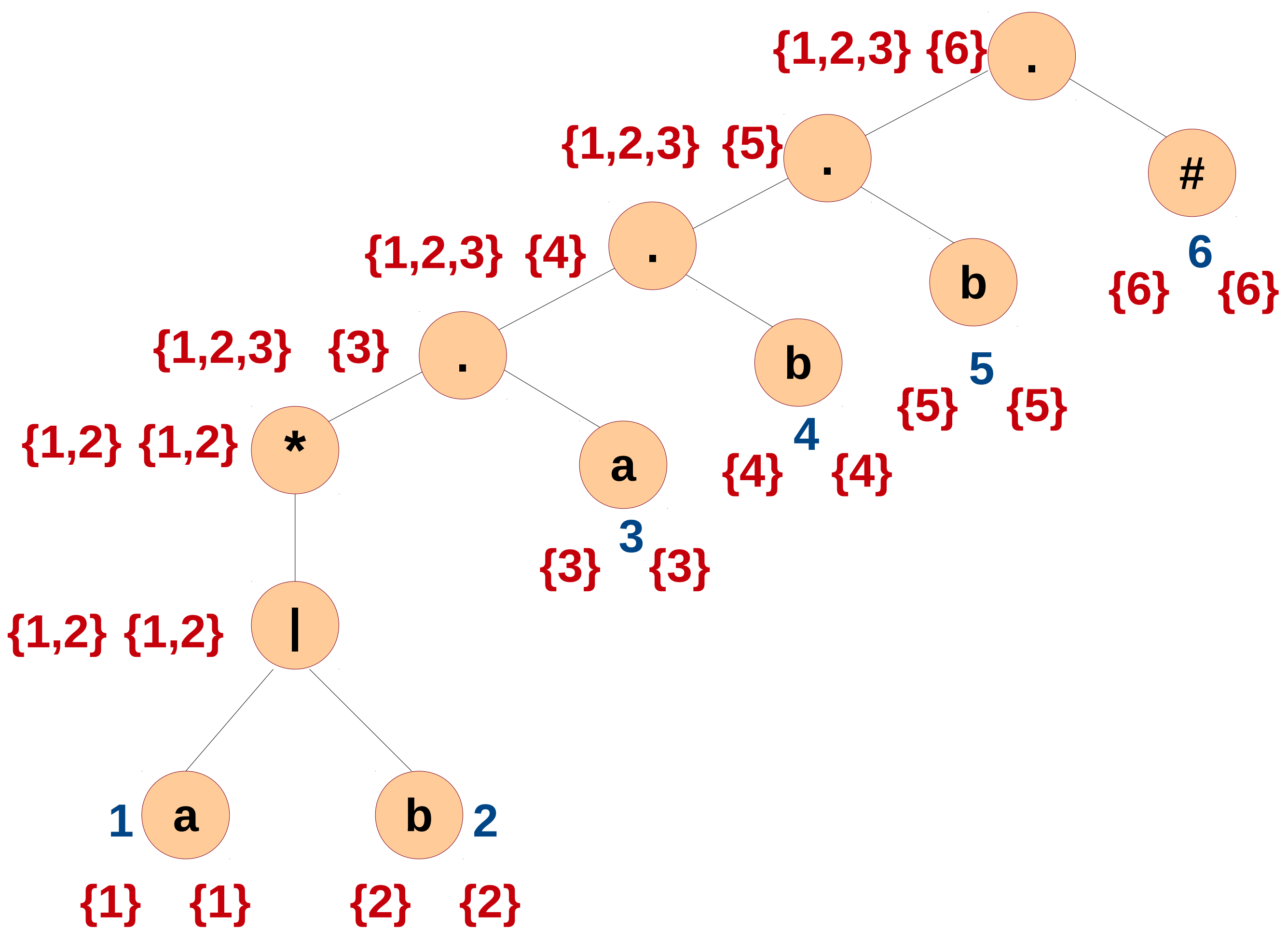
Node n	firstpos(n)
leaf labeled ϵ	{ }
leaf with position i	{i}
or-node n = c1 | c2	firstpos(c1) U firstpos(c2)
cat-node n = c1c2	if (nullable(c1)) firstpos(c1) U firstpos(c2) else firstpos(c1)
star-node n = c*	firstpos(c)

Classwork: Write down the rules for lastpos(n).

lastpos

Node n	**lastpos(n)**
leaf labeled ϵ	{ }
leaf with position i	{i}
or-node n = c1 | c2	lastpos(c1) U lastpos(c2)
cat-node n = c1c2	if (nullable(c2)) lastpos(c1) U lastpos(c2) else lastpos(c2)
star-node n = c*	lastpos(c)

firstpos lastpos

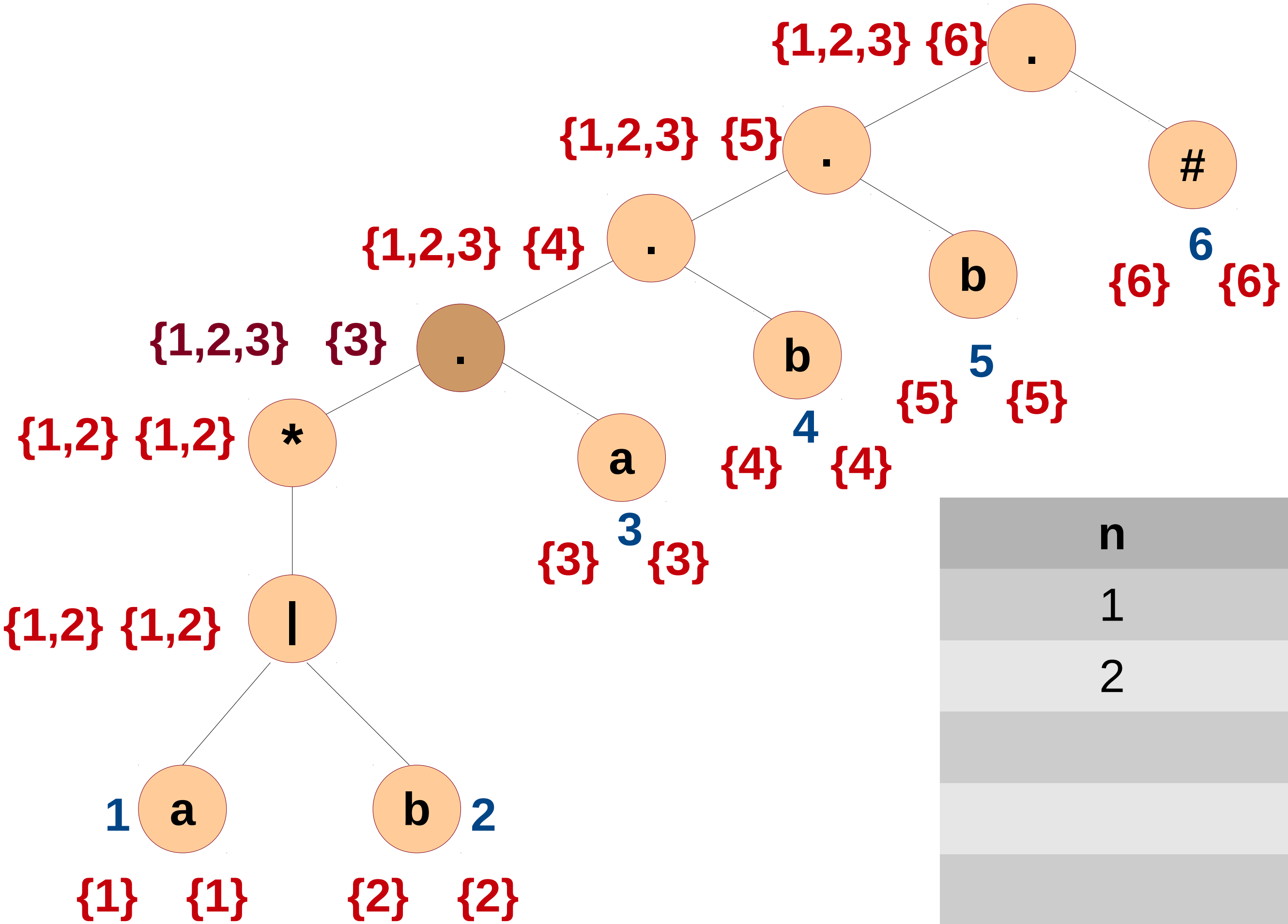


followpos

- *followpos*(n): set of next possible positions from n for valid strings.
  - If n is a **cat-node** with child nodes c1 and c2, then for each position in *lastpos(c1)*, all positions in *firstpos(c2)* *follow*.
  - If n is a **star-node**, then for each position in *lastpos(n)*, all positions in *firstpos(n)* *follow*.

followpos

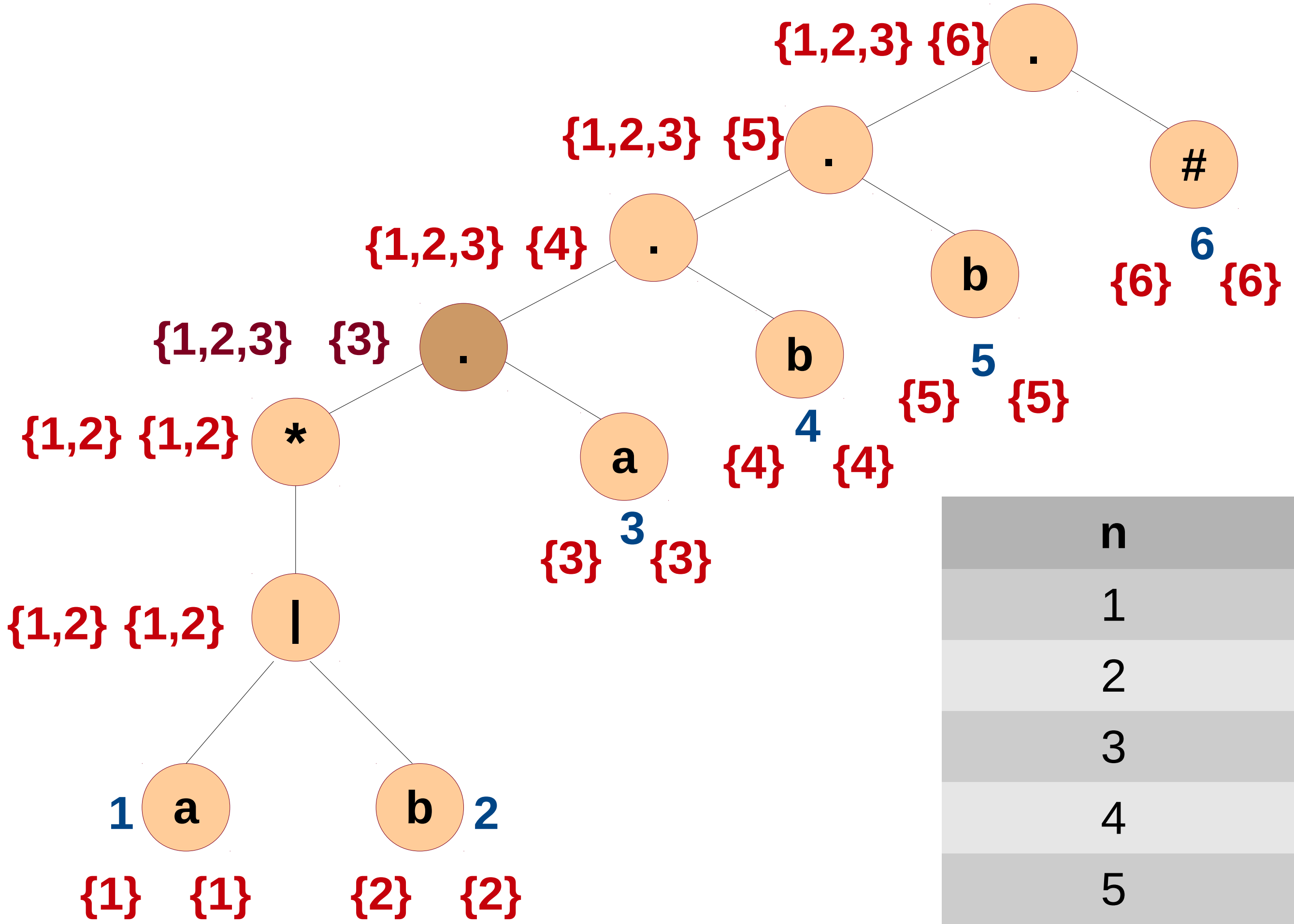
If n is a **cat-node** with child nodes c1 and c2, then for each position in *lastpos(c1)*, all positions in *firstpos(c2)* *follow*.



n	followpos(n)
1	{3}
2	{3}

followpos

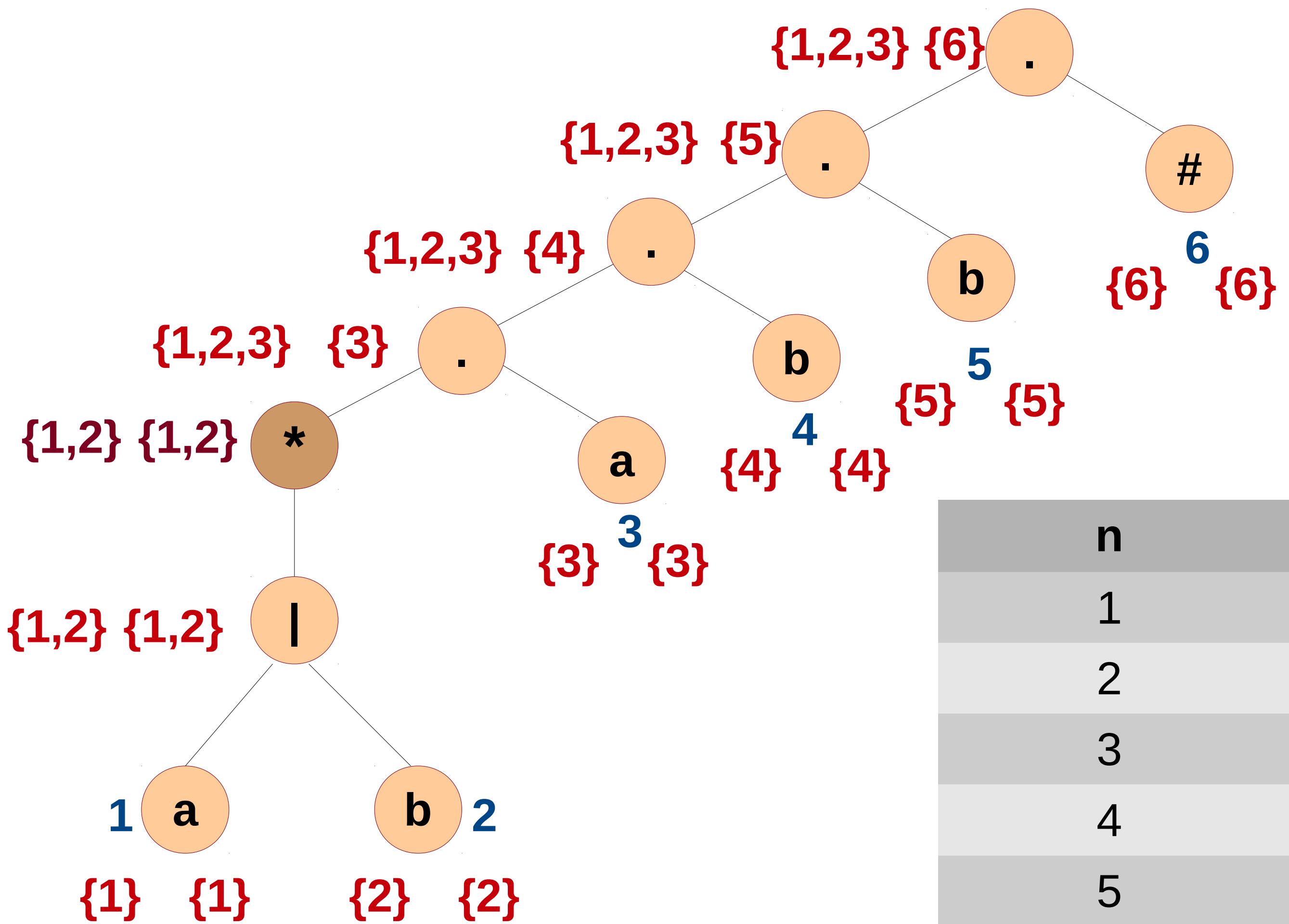
If n is a **cat-node** with child nodes c1 and c2, then for each position in *lastpos(c1)*, all positions in *firstpos(c2)* *follow*.



n	followpos(n)
1	{3}
2	{3}
3	{4}
4	{5}
5	{6}
6	{ }

followpos

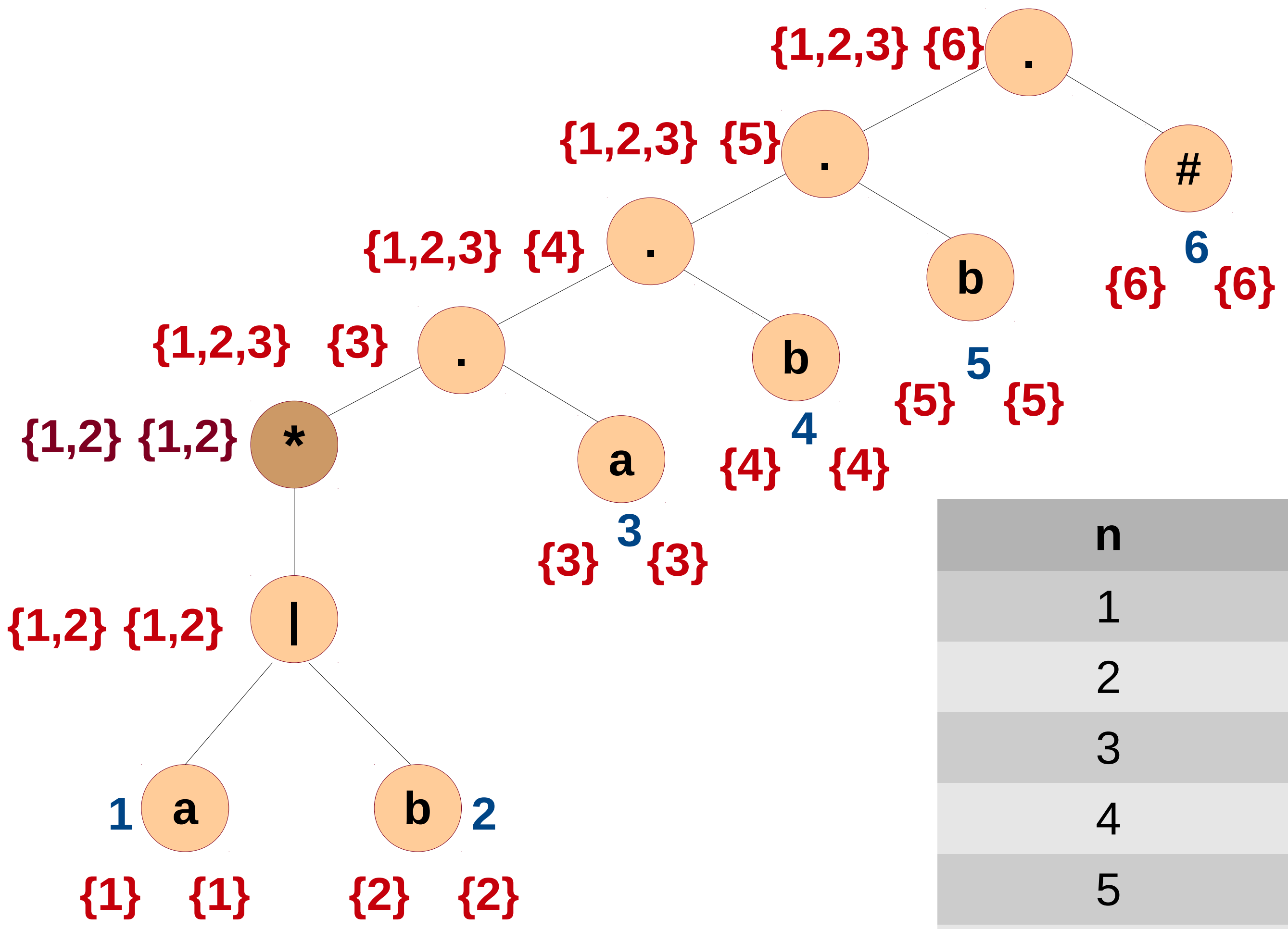
If n is a **star-node**, then for each position in *lastpos(n)*, all positions in *firstpos(n)* *follow*.



n	followpos(n)
1	{3}
2	{3}
3	{4}
4	{5}
5	{6}
6	{ }

followpos

If n is a **star-node**, then for each position in *lastpos(n)*, all positions in *firstpos(n)* *follow*.



n	followpos(n)
1	{3, 1, 2}
2	{3, 1, 2}
3	{4}
4	{5}
5	{6}
6	{ }

Regex → DFA

1. Construct a syntax tree for regex#.
2. Compute *nullable*, *firstpos*, *lastpos*, *followpos*.
3. Construct DFA using transition function *(next slide)*.
4. Mark *firstpos(root)* as start state.
5. Mark states that contain position of # as accepting states.

DFA Transitions

create unmarked state *firstpos(root)*.

while there exists unmarked state s {

    mark s

    for each input symbol *a* {

        uf = U followpos(p) where p is in s labeled *a*

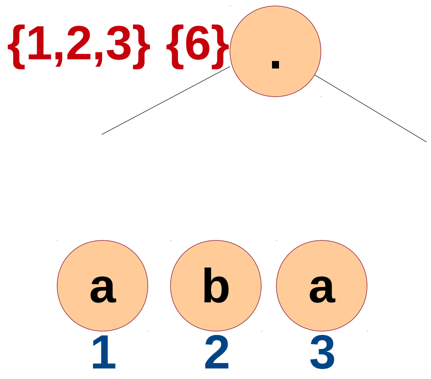
        transition[s, *a*] = uf

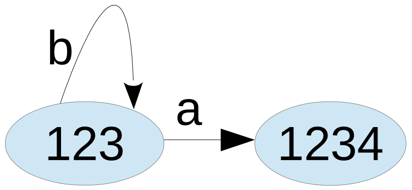
        if uf does not exist

            unmark uf

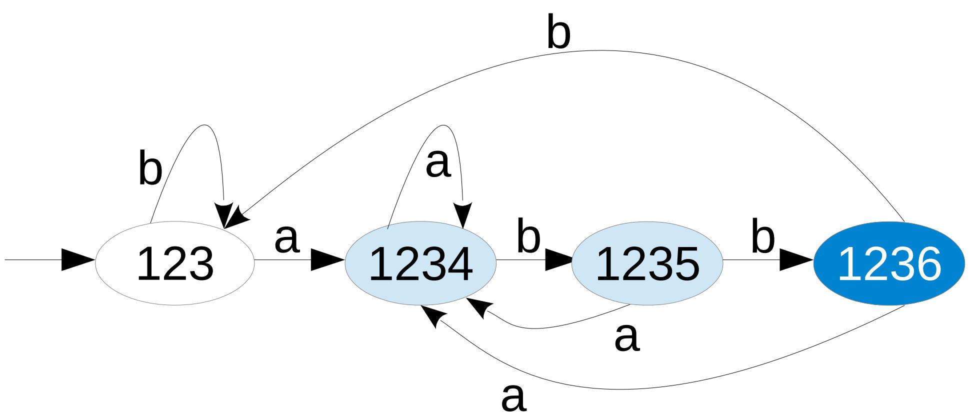
    }

}

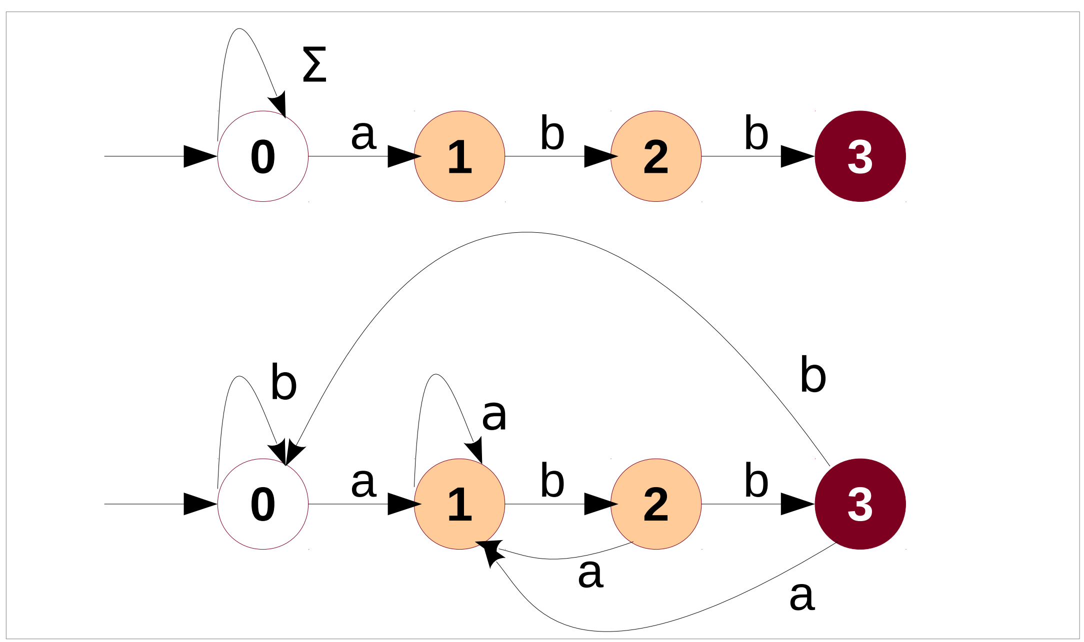




Final DFA



DFA



NFA

DFA

Regex → DFA

1. Construct a syntax tree for regex#.
2. Compute *nullable*, *firstpos*, *lastpos*, *followpos*.
3. Construct DFA using transition function.
4. Mark *firstpos(root)* as start state.
5. Mark states that contain position of # as accepting states.

Do this for (b|ab)*(aa|b)*.

In case you are wondering...

- What to do with this DFA?
  - Recognize strings during lexical analysis.
  - Could be used in utilities such as *grep*.
  - Could be used in regex libraries as supported in php, python, perl, ...

Lexing Summary

- Basic *lex*
- Input Buffering
- KMP String Matching
- Regex → NFA → DFA
- Regex → DFA