Phases of Syntax Analysis

1. Identify the words: **Lexical Analysis**.
   Converts a stream of characters (input program) into a stream of tokens.
   Also called *Scanning* or *Tokenizing*.
2. Identify the sentences: **Parsing**.
   Derive the structure of sentences: construct *parse trees* from a stream of tokens.

Lexical Analysis

Convert a stream of characters into a stream of *tokens*.

- Simplicity: Conventions about "words" are often different from conventions about "sentences".
- Efficiency: Word identification problem has a much more efficient solution than sentence identification problem.
- Portability: Character set, special characters, device features.

Terminology

- Token: Name given to a family of words.
  e.g., integer_constant
- Lexeme: Actual sequence of characters representing a word.
  e.g., 32894
- Pattern: Notation used to identify the set of lexemes represented by a token.
  e.g., $[0-9]+$

Terminology

A few more examples:

Token	Sample Lexemes	Pattern
while	`while`	`while`
integer_constant	$32894, -1093, 0$	$(-\|\epsilon)[0-9]+$
identifier	`buffer_size`	$[\_a-zA-Z]+$

Patterns

How do we *compactly* represent the set of all lexemes corresponding to a token?
For instance:

> The token integer_constant represents the set of all integers: that is, all sequences of digits (0–9), preceded by an optional sign (+ or −).

Obviously, we cannot simply enumerate all lexemes.

Use **Regular Expressions**.

Regular Expressions

Notation to represent (potentially) infinite sets of strings over alphabet Σ.

- a: stands for the set $\{\texttt{a}\}$ that contains a single string a.
- $a \mid b$: stands for the set $\{\texttt{a}, \texttt{b}\}$ that contains two strings a and b.
  - ▷ Analogous to *Union*.
- ab: stands for the set $\{\texttt{ab}\}$ that contains a single string ab.
  - ▷ Analogous to *Product*.
  - ▷ $(a|b)(a|b)$: stands for the set $\{\texttt{aa}, \texttt{ab}, \texttt{ba}, \texttt{bb}\}$.
- a^*: stands for the set $\{\epsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \ldots\}$ that contains all strings of zero or more a's.
  - ▷ Analogous to *closure* of the product operation.

Regular Expressions

Examples of Regular Expressions over $\{\texttt{a}, \texttt{b}\}$:

- $(a|b)^*$: Set of strings with zero or more a's and zero or more b's:
  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$
- (a^*b^*): Set of strings with zero or more a's and zero or more b's such that all a's occur before any b:
  $\{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, \ldots\}$
- $(a^*b^*)^*$: Set of strings with zero or more a's and zero or more b's:
  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

Language of Regular Expressions

Let R be the set of all regular expressions over Σ. Then,

- Empty String: $\epsilon \in R$
- Unit Strings: $\alpha \in \Sigma \Rightarrow \alpha \in R$
- Concatenation: $r_1, r_2 \in R \Rightarrow r_1 r_2 \in R$
- Alternative: $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$
- Kleene Closure: $r \in R \Rightarrow r^* \in R$

Regular Expressions

Example: $(a \mid b)^*$

$$
\begin{aligned}
L_0 &= \{\epsilon\} \\
L_1 &= L_0 \cdot \{\texttt{a}, \texttt{b}\} \\
&= \{\epsilon\} \cdot \{\texttt{a}, \texttt{b}\} \\
&= \{\texttt{a}, \texttt{b}\} \\
L_2 &= L_1 \cdot \{\texttt{a}, \texttt{b}\} \\
&= \{\texttt{a}, \texttt{b}\} \cdot \{\texttt{a}, \texttt{b}\} \\
&= \{\texttt{aa}, \texttt{ab}, \texttt{ba}, \texttt{bb}\} \\
L_3 &= L_2 \cdot \{\texttt{a}, \texttt{b}\} \\
&\vdots \\
L = \bigcup_{i=0}^{\infty} L_i &= \{\epsilon, \texttt{a}, \texttt{b}, \texttt{aa}, \texttt{ab}, \texttt{ba}, \texttt{bb}, \ldots\}
\end{aligned}
$$

Semantics of Regular Expressions

Semantic Function $\mathcal{L}$: Maps regular expressions to sets of strings.

$$
\begin{array}{rcl}
\mathcal{L}(\epsilon) & = & \{\epsilon\} \\
\mathcal{L}(\alpha) & = & \{\alpha\} \qquad (\alpha \in \Sigma) \\
\mathcal{L}(r_1 \mid r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) \\
\mathcal{L}(r_1\ r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) \\
\mathcal{L}(r^*) & = & \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))
\end{array}
$$

Computing the Semantics

$$
\begin{array}{rcl}
\mathcal{L}(a) & = & \{\mathtt{a}\} \\
\mathcal{L}(a \mid b) & = & \mathcal{L}(a) \cup \mathcal{L}(b) \\
 & = & \{\mathtt{a}\} \cup \{\mathtt{b}\} \\
 & = & \{\mathtt{a}, \mathtt{b}\} \\
\mathcal{L}(ab) & = & \mathcal{L}(a) \cdot \mathcal{L}(b) \\
 & = & \{\mathtt{a}\} \cdot \{\mathtt{b}\} \\
 & = & \{\mathtt{ab}\} \\
\mathcal{L}((a \mid b)(a \mid b)) & = & \mathcal{L}(a \mid b) \cdot \mathcal{L}(a \mid b) \\
 & = & \{\mathtt{a}, \mathtt{b}\} \cdot \{\mathtt{a}, \mathtt{b}\} \\
 & = & \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}
\end{array}
$$

Computing the Semantics of Closure

Example: $\mathcal{L}((a \mid b)^*)$
$= \{\epsilon\} \cup (\mathcal{L}(a \mid b) \cdot \mathcal{L}((a \mid b)^*))$

$$
\begin{array}{rcl}
L_0 & = & \{\epsilon\} \qquad\qquad \textit{Base case} \\
L_1 & = & \{\epsilon\} \cup (\{\mathtt{a}, \mathtt{b}\} \cdot L_0) \\
 & = & \{\epsilon\} \cup (\{\mathtt{a}, \mathtt{b}\} \cdot \{\epsilon\}) \\
 & = & \{\epsilon, \mathtt{a}, \mathtt{b}\} \\
L_2 & = & \{\epsilon\} \cup (\{\mathtt{a}, \mathtt{b}\} \cdot L_1) \\
 & = & \{\epsilon\} \cup (\{\mathtt{a}, \mathtt{b}\} \cdot \{\epsilon, \mathtt{a}, \mathtt{b}\}) \\
 & = & \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\} \\
\vdots & &
\end{array}
$$

$$\mathcal{L}((a \mid b)^*) = L_\infty = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \ldots\}$$

Another Example

$\boldsymbol{\mathcal{L}((a^*b^*)^*)}$:

$$
\begin{array}{rcl}
\mathcal{L}(a^*) & = & \{\epsilon, \mathtt{a}, \mathtt{aa}, \ldots\} \\
\mathcal{L}(b^*) & = & \{\epsilon, \mathtt{b}, \mathtt{bb}, \ldots\} \\
\mathcal{L}(a^*b^*) & = & \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{bb}, \\
 & & \quad \mathtt{aaa}, \mathtt{aab}, \mathtt{abb}, \mathtt{bbb}, \ldots\} \\
\mathcal{L}((a^*b^*)^*) & = & \{\epsilon\} \\
 & & \cup\{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{bb}, \\
 & & \quad \mathtt{aaa}, \mathtt{aab}, \mathtt{abb}, \mathtt{bbb}, \ldots\} \\
 & & \cup\{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \\
 & & \quad \mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \mathtt{abb}, \mathtt{baa}, \mathtt{bab}, \mathtt{bba}, \mathtt{bbb}, \ldots\} \\
 & & \vdots \\
 & = & \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \ldots\}
\end{array}
$$

Regular Definitions

Assign "names" to regular expressions.
For example,

$$
\begin{array}{rcl}
\mathsf{digit} & \longrightarrow & 0 \mid 1 \mid \cdots \mid 9 \\
\mathsf{natural} & \longrightarrow & \mathsf{digit}\ \mathsf{digit}^*
\end{array}
$$

SHORTHANDS:

- a^+: Set of strings with one or more occurrences of a.
- $a^?$: Set of strings with zero or one occurrences of a.

Example:

$$
\mathsf{integer} \longrightarrow (+|-)^? \mathsf{digit}^+
$$

Regular Definitions: Examples

$$
\begin{array}{rcl}
\mathsf{float} & \longrightarrow & \mathsf{integer}\ .\ \mathsf{fraction} \\
\mathsf{integer} & \longrightarrow & (+|-)^?\ \mathsf{no\_leading\_zero} \\
\mathsf{no\_leading\_zero} & \longrightarrow & (\mathsf{nonzero\_digit}\ \mathsf{digit}^*) \mid 0 \\
\mathsf{fraction} & \longrightarrow & \mathsf{no\_trailing\_zero}\ \mathsf{exponent}^? \\
\mathsf{no\_trailing\_zero} & \longrightarrow & (\mathsf{digit}^*\ \mathsf{nonzero\_digit}) \mid 0 \\
\mathsf{exponent} & \longrightarrow & (\mathtt{E} \mid \mathtt{e})\ \mathsf{integer} \\
\mathsf{digit} & \longrightarrow & 0 \mid 1 \mid \cdots \mid 9 \\
\mathsf{nonzero\_digit} & \longrightarrow & 1 \mid 2 \mid \cdots \mid 9
\end{array}
$$

Regular Definitions and Lexical Analysis

Regular Expressions and Definitions *specify* sets of strings over an input alphabet.

- They can hence be used to specify the set of *lexemes* associated with a *token*.
  - ▷ Used as the *pattern* language

How do we decide whether an input string belongs to the set of strings specified by a regular expression?

Using Regular Definitions for Lexical Analysis

Q: Is ababbaabbb in $\mathcal{L}(((a^*b^*)^*)$?
A: Hm. Well. Let's see.

$$\begin{aligned}\mathcal{L}((a^*b^*)^*) &= \{\epsilon\} \\ &\cup\{\epsilon, \texttt{a}, \texttt{b}, \texttt{aa}, \texttt{ab}, \texttt{bb}, \\ &\quad \texttt{aaa}, \texttt{aab}, \texttt{abb}, \texttt{bbb}, \ldots\} \\ &\cup\{\epsilon, \texttt{a}, \texttt{b}, \texttt{aa}, \texttt{ab}, \texttt{ba}, \texttt{bb}, \\ &\quad \texttt{aaa}, \texttt{aab}, \texttt{aba}, \texttt{abb}, \texttt{baa}, \texttt{bab}, \texttt{bba}, \texttt{bbb}, \ldots\} \\ &\vdots \\ &= ???\end{aligned}$$

Lexical Analysis

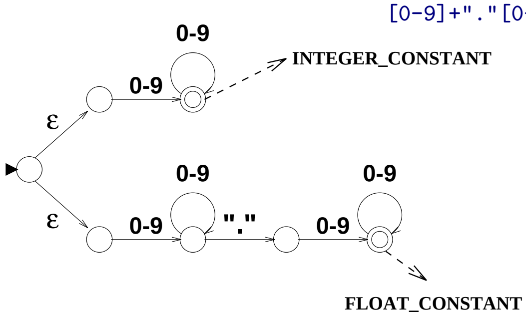
- Regular Expressions and Definitions are used to specify the set of strings (lexemes) corresponding to a *token*.
- An automaton (DFA/NFA) is built from the above specifications.
- Each final state is associated with an *action*: emit the corresponding token.

Specifying Lexical Analysis

Consider a recognizer for integers (sequence of digits) and floats (sequence of digits separated by a decimal point).

```
[0-9]+              { emit(INTEGER_CONSTANT); }

[0-9]+"."[0-9]+     { emit(FLOAT_CONSTANT); }
```



Lex

Tool for building lexical analyzers.
Input: lexical specifications (`.l` file)
Output: C function (`yylex`) that returns a token on each invocation.

```
%%
[0-9]+              { return(INTEGER_CONSTANT); }

[0-9]+"."[0-9]+     { return(FLOAT_CONSTANT); }
```

Tokens are simply integers (`#define`'s).

Lex Specifications

```
%{
    C header statements for inclusion
%}
  Regular Definitions    e.g.:
    digit    [0-9]
%%

  Token Specifications    e.g.:
    {digit}+              { return(INTEGER_CONSTANT); }

%%
  Support functions in C
```

Regular Expressions in Lex

Adds "syntactic sugar" to regular expressions:

- Range: `[0-7]`: Integers from 0 through 7 (inclusive)
  `[a-nx-zA-Q]`: Letters `a` thru `n`, `x` thru `z` and `A` thru `Q`.
- Exception: `[^/]`: Any character other than `/`.
- Definition: `{digit}`: Use the previously specified regular definition `digit`.
- Special characters: Connectives of regular expression, convenience features.
  e.g.: `| * ^`

Special Characters in Lex

`| * + ? ( )`	Same as in regular expressions
`[ ]`	Enclose ranges and exceptions
`{ }`	Enclose "names" of regular definitions
`^`	Used to negate a specified range (in Exception)
`.`	Match any single character except newline
`\`	Escape the next character
`\n`, `\t`	Newline and Tab

For literal matching, enclose special characters in double quotes (`"`) *e.g.:* `"*"`
Or use `\` to escape. *e.g.:* `\"`

Examples

`for`	Sequence of `f`, `o`, `r`
`"||"`	C-style OR operator (two vert. bars)
`.*`	Sequence of non-newline characters
`[^*/]+`	Sequence of characters except `*` and `/`
`\"[^"]*\"`	Sequence of non-quote characters beginning and ending with a quote
`({letter}|"_")({letter}|{digit}|"_")*`	C-style identifiers

A Complete Example

```
%{
#include <stdio.h>
#include "tokens.h"
%}
digit  [0-9]
hexdigit  [0-9a-f]
%%

"+"                     { return(PLUS); }
"-"                     { return(MINUS); }
{digit}+                { return(INTEGER_CONSTANT); }
{digit}+"."{digit}+     { return(FLOAT_CONSTANT); }
.                       { return(SYNTAX_ERROR); }
%%
```

Actions

Actions are attached to final states.

- Distinguish the different final states.
- Used to return *tokens*.
- Can be used to set *attribute values*.
- Fragment of C code (blocks enclosed by '{' and '}').

Attributes

Additional information about a token's lexeme.

- Stored in variable `yylval`
- Type of attributes (usually a union) specified by `YYSTYPE`
- Additional variables:
  - `yytext`: Lexeme (*Actual text string*)
  - `yyleng`: length of string in `yytext`
  - ▷ `yylineno`: Current line number (number of '`\n`' seen thus far)
    - ∗ enabled by `%option yylineno`

Priority of matching

What if an input string matches more than one pattern?

```
"if"                    { return(TOKEN_IF); }
{letter}+               { return(TOKEN_ID); }
"while"                 { return(TOKEN_WHILE); }
```

- A pattern that matches the longest string is chosen.
  Example: `if1` is matched with an identifier, not the keyword `if`.
- Of patterns that match strings of same length, the first (from the top of file) is chosen.
  Example: `while` is matched as an identifier, not the keyword `while`.

Constructing Scanners using (f)lex

- Scanner specifications: *specifications*.`l`

  specifications.`l` $\xrightarrow{\texttt{(f)lex}}$ `lex.yy.c`

- Generated scanner in `lex.yy.c`

  `lex.yy.c` $\xrightarrow{\texttt{(g)cc}}$ *executable*

  – `yywrap()`: hook for signalling end of file.
  – Use `-lfl` (flex) or `-ll` (lex) flags at link time to include default function `yywrap()` that always returns 1.

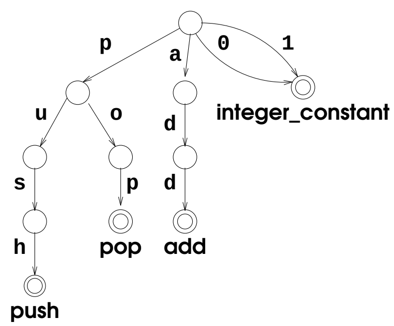
Recognizers

Construct *automata* that recognize strings belonging to a language.

- Finite State Automata ⇒ Regular Languages
  - ▷ Finite State → cannot maintain arbitrary counts.
- Push Down Automata ⇒ Context-free Languages
  - ▷ Stack is used to maintain counter, but only one counter can go arbitrarily high.

Recognizing Finite Sets of Strings

Identifying words from a small, finite, fixed vocabulary is straightforward.
For instance, consider a stack machine with `push`, `pop`, and `add` operations with two constants: `0` and `1`.
We can use the *automaton*:



Finite State Automata

Represented by a labeled directed graph.

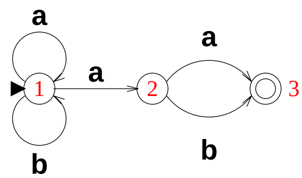
- A finite set of *states* (vertices).
- *Transitions* between states (edges).
- *Labels* on transitions are drawn from $\Sigma \cup \{\epsilon\}$.
- One distinguished *start* state.
- One or more distinguished *final* states.

Finite State Automata: An Example

Consider the Regular Expression $(a \mid b)^*a(a \mid b)$.

$\mathcal{L}((a \mid b)^*a(a \mid b)) = \{$aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, ...$\}$.

The following automaton determines whether an input string belongs to $\mathcal{L}((a \mid b)^*a(a \mid b)$:



Using States in Lex

- Some regular languages are more easily expressed as FSA
  - Set of all strings representing binary numbers divisible by 3
- Lex allows you to use FSA concepts using *start states*

```
%x MOD1 MOD2
"0" { }
"1" {BEGIN MOD1}
<MOD1> "0"  {BEGIN MOD2}
<MOD1> "1"  {BEGIN 0}
```

Other Special Directives

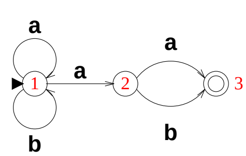
- `ECHO` causes Lex to echo current lexeme
- `REJECT` causes abandonment of current match in favor of the next.
- Example

```
a|
ab|
abc|
abcd {ECHO; REJECT;}
.|\n {/* eat up the character */}
```

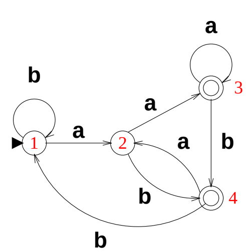
Deterministic Vs Nondeterministic FSA

$(a \mid b)^*a(a \mid b)$:

Nondeterministic:
(NFA)



Deterministic:
(DFA)



Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string x

... if beginning from the start state

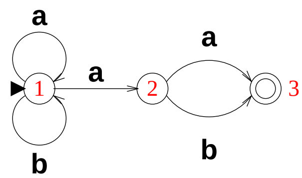
... we can trace some path through the automaton

... such that the sequence of edge labels spells x

... and end in a final state.

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?

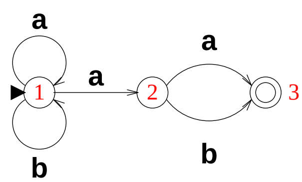


Input:		a	b	a	b
Path 1:	1				
Path 2:					
Path 3:					

Accept

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?

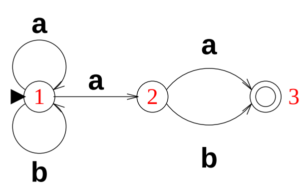


Input:		a	b	a	b
Path 1:	1	1			
Path 2:					
Path 3:					

Accept

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?

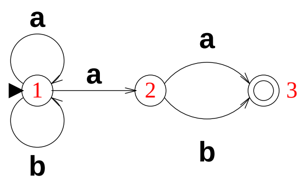


Input:		a	b	a	b
Path 1:	1	1	1		
Path 2:					
Path 3:					

Accept

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?

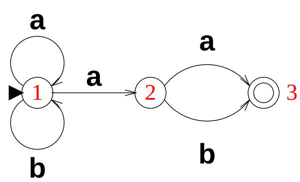


Input:		a	b	a	b
Path 1:	1	1	1	1	1
Path 2:					
Path 3:					

Accept

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?

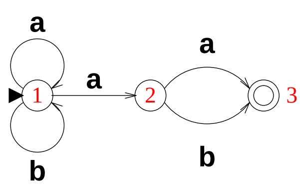


Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:						

Accept

Recognition with an NFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?

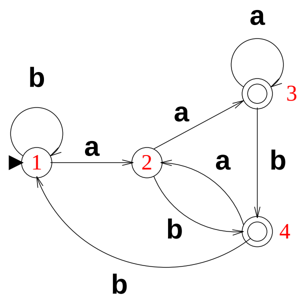


Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	

Accept

Recognition with a DFA

Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b	
Path:	1	2	4	2	4	Accept

NFA vs. DFA

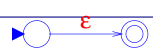
For every NFA, there is a DFA that accepts the same set of strings.

- NFA may have transitions labeled by ϵ.
  (Spontaneous transitions)
- All transition labels in a DFA belong to Σ.
- For some string x, there may be *many* accepting paths in an NFA.
- For all strings x, there is *one unique* accepting path in a DFA.
- Usually, an input string can be recognized *faster* with a DFA.
- NFAs are typically *smaller* than the corresponding DFAs.

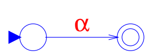
Regular Expressions to NFA

Thompson's Construction: For every regular expression r, derive an NFA $N(r)$ with unique start and final states.

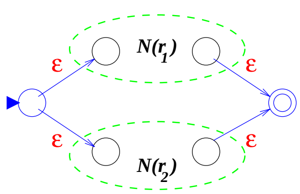
ϵ



$\alpha \in \Sigma$

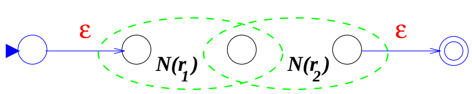


$(r_1 \mid r_2)$

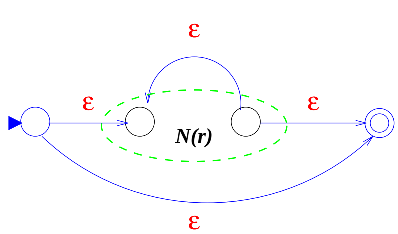


Regular Expressions to NFA (contd.)

$r_1 r_2$

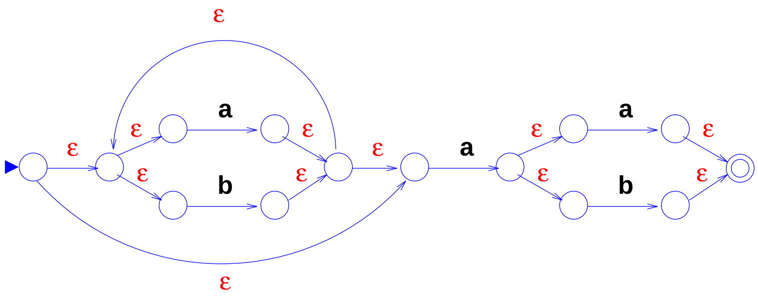


r^*



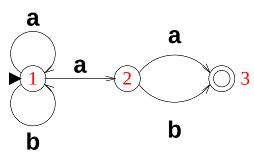
Example

$(a \mid b)^* a(a \mid b)$:



Recognition with an NFA

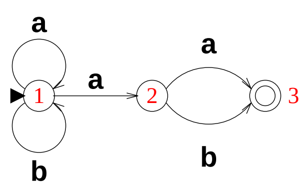
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	b	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,3\}$	$\{1,2\}$	$\{1,3\}$	Accept

Recognition with an NFA (contd.)

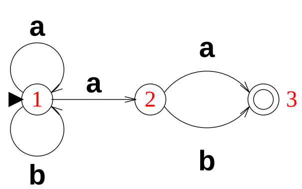
Is aaab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	a	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	1	2	
Path 3:	1	1	1	2	3	Accept
Path 4:	1	1	2	3	$\perp$	
Path 5:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,2,3\}$	Accept

Recognition with an NFA (contd.)

Is aabb $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	a	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	2	3	$\perp$	
Path 3:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,3\}$	$\{1\}$	REJECT

Converting NFA to DFA

Subset construction

Given a set S of NFA states,

- compute $S_\epsilon = \epsilon\text{-closure}(S)$: S_ϵ is the set of all NFA states reachable by zero or more ϵ-transitions from S.

- compute $S_\alpha = \mathsf{goto}(S, \alpha)$:
  - S' is the set of all NFA states reachable from S by taking a transition labeled α.
  - $S_\alpha = \epsilon\text{-closure}(S')$.

Converting NFA to DFA (contd).

Each state in DFA corresponds to a *set of states* in NFA.
Start state of DFA = ϵ-closure(start state of NFA).
From a state s in DFA that corresponds to a set of states S in NFA:

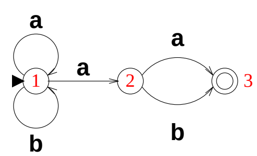
add a transition labeled α to state s' that corresponds to a non-empty S' in NFA,

such that $S' = \text{goto}(S, \alpha)$.

s is a state in DFA such that the corresponding set of states S in NFA contains a final state of NFA,

$\Leftarrow$ s is a final state of DFA

NFA $\rightarrow$ DFA: An Example



ϵ-closure($\{1\}$)	=	$\{1\}$
goto($\{1\}, \texttt{a}$)	=	$\{1, 2\}$
goto($\{1\}, \texttt{b}$)	=	$\{1\}$
goto($\{1, 2\}, \texttt{a}$)	=	$\{1, 2, 3\}$
goto($\{1, 2\}, \texttt{b}$)	=	$\{1, 3\}$
goto($\{1, 2, 3\}, \texttt{a}$)	=	$\{1, 2, 3\}$

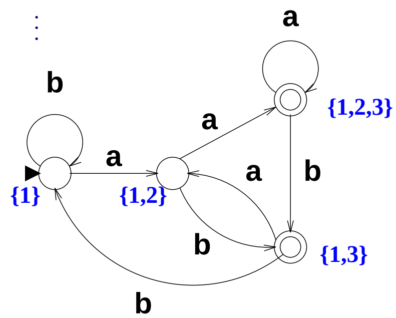
$\vdots$

NFA $\rightarrow$ DFA: An Example (contd.)

ϵ-closure($\{1\}$)	=	$\{1\}$
goto($\{1\}, \texttt{a}$)	=	$\{1, 2\}$
goto($\{1\}, \texttt{b}$)	=	$\{1\}$
goto($\{1, 2\}, \texttt{a}$)	=	$\underline{\{1, 2, 3\}}$
goto($\{1, 2\}, \texttt{b}$)	=	$\underline{\{1, 3\}}$
goto($\{1, 2, 3\}, \texttt{a}$)	=	$\underline{\{1, 2, 3\}}$
goto($\{1, 2, 3\}, \texttt{b}$)	=	$\{1\}$
goto($\{1, 3\}, \texttt{a}$)	=	$\{1, 2\}$
goto($\{1, 3\}, \texttt{b}$)	=	$\{1\}$

NFA $\rightarrow$ DFA: An Example (contd.)

$$
\begin{aligned}
\text{goto}(\{1\}, \texttt{a}) &= \{1, 2\} \\
\text{goto}(\{1\}, \texttt{b}) &= \{1\} \\
\text{goto}(\{1, 2\}, \texttt{a}) &= \{1, 2, 3\} \\
\text{goto}(\{1, 2\}, \texttt{b}) &= \{1, 3\} \\
\text{goto}(\{1, 2, 3\}, \texttt{a}) &= \{1, 2, 3\} \\
&\vdots
\end{aligned}
$$



NFA vs. DFA

R = Size of Regular Expression
N = Length of Input String

	NFA	DFA
Size of Automaton	$O(R)$	$O(2^R)$
Recognition time per input string	$O(N \times R)$	$O(N)$

Implementing a Scanner

```
transition : state × Σ → state

algorithm scanner() {
    current_state = start state;
    while (1) {
        c = getc(); /* on end of file, ... */
        if defined(transition(current_state, c))
            current_state = transition(current_state, c);
        else
            return s;
    }
}
```

Implementing a Scanner (contd.)

Implementing the *transition* function:

- Simplest: 2-D array.
  Space inefficient.
- Traditionally compressed using row/colum equivalence. (default on `(f)lex`)
  Good space-time tradeoff.
- Further table compression using various techniques:

– Example: RDM (Row Displacement Method):
  Store rows in overlapping manner using 2 1-D arrays.

Smaller tables, but longer access times.

Lexical Analysis: A Summary

Convert a stream of characters into a stream of tokens.

- Make rest of compiler independent of character set
- Strip off comments
- Recognize line numbers
- Ignore white space characters
- Process macros (definitions and uses)
- Interface with **symbol** (name) **table**.