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 over alphabet Σ

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: $r \in R \Rightarrow r^* \in R$

Semantics of Regular Expressions

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

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

set union

set product

$\mathcal{L}(r) \cdot \mathcal{L}(r)$

$\mathcal{L}(r) \cdot \mathcal{L}(r) \cdot \mathcal{L}(r)$

Computing the Semantics

$$
\begin{aligned}
\mathcal{L}(a) &= \{\mathrm{a}\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cup \{\mathrm{b}\} \\
&= \{\mathrm{a}, \mathrm{b}\}
\end{aligned}
$$

Computing the Semantics

$$
\begin{aligned}
\mathcal{L}(a) &= \{\mathrm{a}\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cup \{\mathrm{b}\} \\
&= \{\mathrm{a}, \mathrm{b}\} \\
\mathcal{L}(ab) &= \mathcal{L}(a) \cdot \mathcal{L}(b) \\
&= \{\mathrm{a}\} \cdot \{\mathrm{b}\} \\
&= \{\mathrm{ab}\}
\end{aligned}
$$

Computing the Semantics

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

Computing the Semantics of Closure

$$\mathcal{L}(r^*) = \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))$$

$\mathcal{L}_0(r^*) = \mathcal{L}(\epsilon)$

$\mathcal{L}_1(r^*) = \mathcal{L}_0 \cup \mathcal{L}(r) \cdot \mathcal{L}_0(r^*)$

$\mathcal{L}_2(r^*) = \mathcal{L}_1(r^*) \cup \mathcal{L}(r) \cdot \mathcal{L}_1(r^*)$

$\mathcal{L}_3(r^*) = \mathcal{L}_2(r^*) \cup \mathcal{L}(r) \cdot \mathcal{L}_2(r^*)$

each $\mathcal{L}_i$ is a closer approximation to $\mathcal{L}$

Computing the Semantics of Closure

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

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

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

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

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

$(a^* b^*)^*$

$(a^* b^*)(a^* b^*)$

abb

bbb

all b's appear after all a's.

Regular Definitions

Assign "names" to regular expressions.

For example,

$$\begin{aligned} \text{digit} &\longrightarrow 0 \mid 1 \mid \cdots \mid 9 \\ \text{natural} &\longrightarrow \text{digit}\ \text{digit}^* \end{aligned}$$

integer

SHORTHANDS:

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

$r+ \equiv r \cdot r^*$

$r? \equiv \epsilon \mid r$

Example:

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

$(\epsilon \mid (+ \mid -))((0|1|2|\ldots|9)(0|1|2|\ldots 9)^*)$

Regular Definitions: Examples

float	$\longrightarrow$	integer . fraction
integer	$\longrightarrow$	$(+\mid-)^?$ no_leading_zero
no_leading_zero	$\longrightarrow$	(nonzero_digit digit*) | 0
fraction	$\longrightarrow$	no_trailing_zero exponent$^?$
no_trailing_zero	$\longrightarrow$	(digit* nonzero_digit) | 0
exponent	$\longrightarrow$	(E | e) integer
digit	$\longrightarrow$	$0 \mid 1 \mid \cdots \mid 9$
nonzero_digit	$\longrightarrow$	$1 \mid 2 \mid \cdots \mid 9$

01
0
1
14
014

5 X
5.0

5.e5 X

1.40

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?

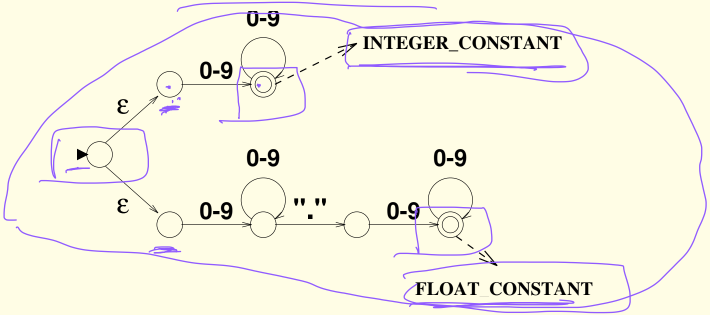
Lexical Analysis

- 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.

Lesk

GNU → Flex

Vern Paxson

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

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

%%
  Support functions in C
```

Definitions

Rules

Proper parts of Flex file

Regular Expressions in Lex

+, ?

Adds "syntactic sugar" to regular expressions:

[;-7]

- Range: `[0-7]`: Integers from 0 through 7 (inclusive)

  `[a-nx-zA-Q]`: Letters a thru n, x thru z and A thru Q.

[-0-9]

[ab^]

- Exception: `[^/]`: Any character other than `/`.
- Definition: `{digit}`: Use the previously specified regular definition `digit`.
- Special characters: Connectives of regular expression, convenience features.

  e.g.: `|` `*` `^`

a, b, ^ [^ab]

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

POSIX

PERL-compatible

PCRE

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); }
%%
```

Syntax analyzer calls yylex

return value

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.

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? Disambiguation

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

- A pattern that matches the longest string is chosen.

  Example: `ifs` is matched with an identifier, not the keyword `if`.

  if 1

- Of patterns that match strings of same length, the first (from the top of file) is chosen.

  while "if b"

  - `while` is matched as an identifier, not the keyword `while`.
  - Given `if1`, a match will be announced for the keyword `if`, with `1` being considered as part of the next token.

Constructing Scanners using `(f)lex`

- Scanner specifications: *specifications*.`l`

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

- Generated scanner in `lex.yy.c`

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

  - `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 $\Rightarrow$ Regular Languages
  - Finite State $\rightarrow$ cannot maintain arbitrary counts.
- Push Down Automata $\Rightarrow$ Context-free Languages
  - Stack is used to maintain counter, but only one counter can go arbitrarily high.

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 $(\boldsymbol{a} \mid \boldsymbol{b})^*\boldsymbol{a}(\boldsymbol{a} \mid \boldsymbol{b})$.

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

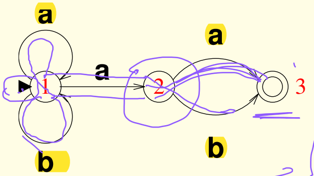
Finite State Automata: An Example

$\Sigma = \{a, b\}$

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

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

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

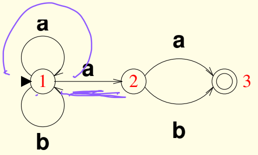


$(a|b)^*aa$
$(a|b)^*ab$

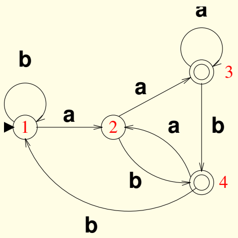
Deterministic Vs Nondeterministic FSA

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

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.

Or, there exists a path in the graph from the start state to a final state such that the sequence of labels on the path spells out x

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.

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	$O(N \times R)$	$O(N)$

+

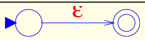
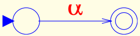
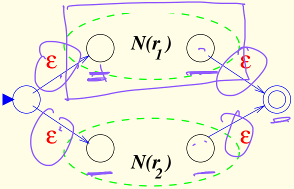
Total cost $O(NR)$ $O(2^R + N)$

$\max(2^R, N)$

Regular Expressions to NFA

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

Base

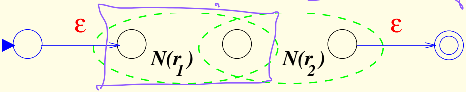
ϵ	
$\alpha \in \Sigma$	
$(r_1 \mid r_2)$	

Regular Expressions to NFA (contd.)

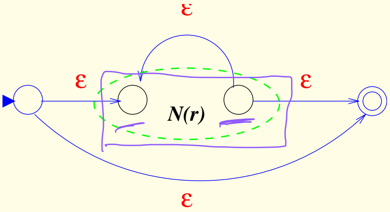
final of r_1

$\bigcirc \xrightarrow{\epsilon} \bigcirc$ start of r_2

$r_1 r_2$

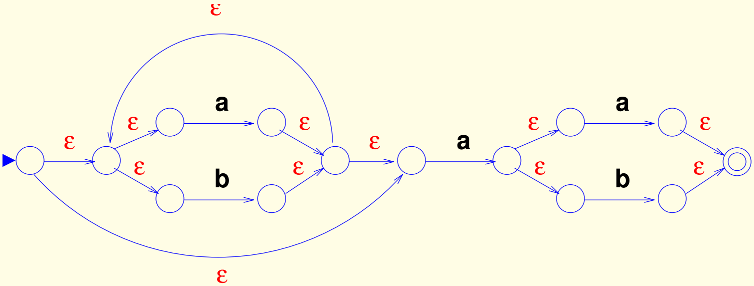


r^*



Example

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



Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs

Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs

Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs
- *Implication:* REs and NFAs have the same expressive power

Expressive Power of RE Vs FSA

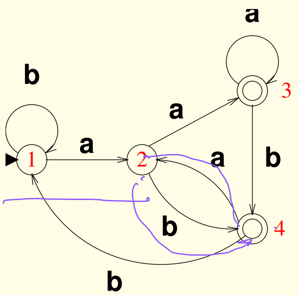
- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs
- *Implication:* REs and NFAs have the same expressive power
- Where do DFAs stand?
  - Every DFA is an NFA
  - We will show that every NFA can be converted into an equivalent DFA

Expressive Power of RE Vs FSA

- We just saw that every RE can be converted into an equivalent NFA
  - Implication: NFAs are at least as expressive as REs
- It can also be shown that every NFA can be converted into an equivalent RE
  - Implication: REs are at least as expressive as NFAs
- *Implication:* REs and NFAs have the same expressive power
- Where do DFAs stand?
  - Every DFA is an NFA
  - We will show that every NFA can be converted into an equivalent DFA
- **Implication:** RE, NFA and DFA are equivalent

Recognition with a DFA

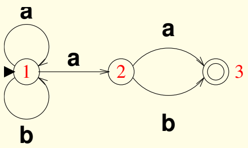
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



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

Recognition with an NFA

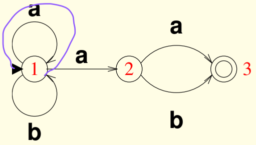
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b
Path 1:	1				

Recognition with an NFA

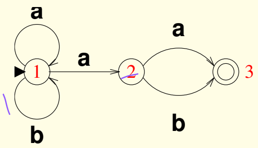
Is abab $\in \mathcal{L}((\boldsymbol{a} \mid \boldsymbol{b})^* \boldsymbol{a}(\boldsymbol{a} \mid \boldsymbol{b}))$?



Input:		a	b	a	b
Path 1:	1	1			

Recognition with an NFA

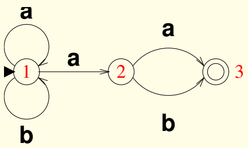
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1		

Recognition with an NFA

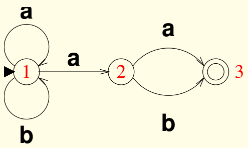
Is abab $\in \mathcal{L}((a \mid b)^*a(a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1	1	

Recognition with an NFA

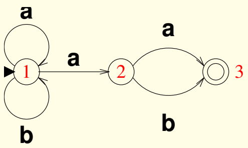
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



Input:		a	b	a	b
Path 1:	1	1	1	1	1

Recognition with an NFA

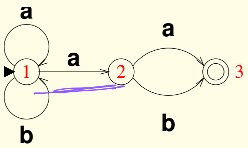
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



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

Recognition with an NFA

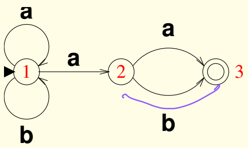
Is abab $\in \mathcal{L}((a \mid b)^* a(a \mid b))$?



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

Recognition with an NFA

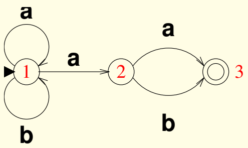
Is abab $\in \mathcal{L}((a \mid b)^* a (a \mid b))$?



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

Recognition with an NFA

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



2^N

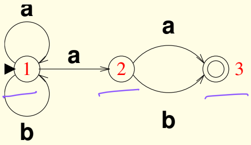
$N \times R$

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	$\perp$	$\perp$	

bottom

Recognition with an NFA

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



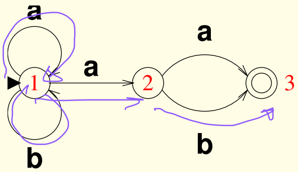
a

$|R|$

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



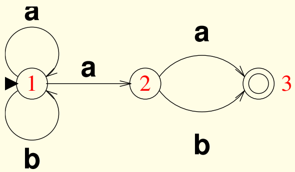
O(R)

O(NR)

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

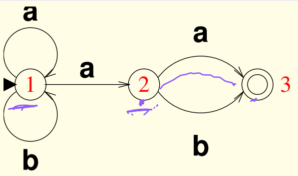
Recognition with an NFA (contd.)

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

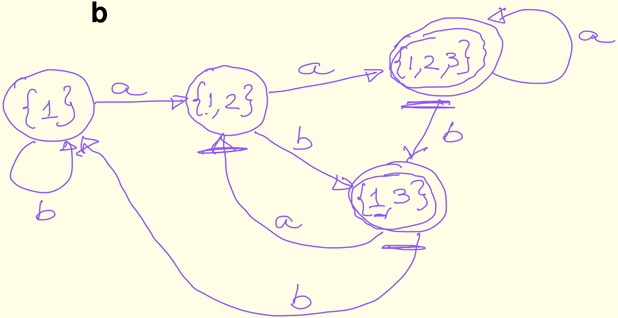


Input:		a	a	a	b	
Path 1:	1	1	1	1	1	
Path 2:	1	1	2	3	$\perp$	
Path 3:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,3\}$	$\{1\}$	REJECT

Converting NFA to DFA



NFA S DFA $\mathcal{P}(S)$



Converting NFA to DFA (contd.)

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 $\boxed{S_\alpha}$ = goto(S, α):
  - 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:

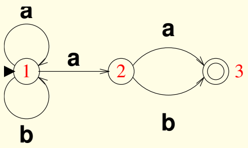
add a transition labeled α to state s' that corresponds to a non-empty S' in NFA,

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 → DFA: An Example



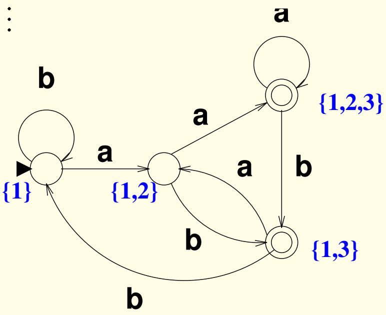
$$
\begin{aligned}
\epsilon\text{-closure}(\{1\}) &= \{1\} \\
\text{goto}(\{1\}, a) &= \{1, 2\} \\
\text{goto}(\{1\}, b) &= \{1\} \\
\text{goto}(\{1, 2\}, a) &= \{1, 2, 3\} \\
\text{goto}(\{1, 2\}, b) &= \{1, 3\} \\
\text{goto}(\{1, 2, 3\}, a) &= \{1, 2, 3\} \\
&\vdots
\end{aligned}
$$

NFA → DFA: An Example (contd.)

$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \{1\} \\
\text{goto}(\{1\}, a) & = & \{1, 2\} \\
\text{goto}(\{1\}, b) & = & \{1\} \\
\text{goto}(\{1, 2\}, a) & = & \underline{\{1, 2, 3\}} \\
\text{goto}(\{1, 2\}, b) & = & \underline{\{1, 3\}} \\
\text{goto}(\{1, 2, 3\}, a) & = & \underline{\{1, 2, 3\}} \\
\text{goto}(\{1, 2, 3\}, b) & = & \{1\} \\
\text{goto}(\{1, 3\}, a) & = & \{1, 2\} \\
\text{goto}(\{1, 3\}, b) & = & \{1\}
\end{array}
$$

NFA → DFA: An Example (contd.)

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



Converting RE to FSA

NFA: Compile RE to NFA (Thompson's construction [1968]), then match.

DFA: Compile to DFA, then match

(A) Convert NFA to DFA (Rabin-Scott construction), minimize

(B) Direct construction: RE derivatives [Brzozowski 1964].

- More convenient and a bit more general than (A).

(C) Direct construction of [McNaughton Yamada 1960]

- Can be seen as a (more easily implemented) specialization of (B).
- Used in Lex and its derivatives, i.e., most compilers use this algorithm.

Converting RE to FSA

- NFA approach takes $O(n)$ NFA construction plus $O(nm)$ matching, so has worst case $O(nm)$ complexity.
- DFA approach takes $O(2^n)$ construction plus $O(m)$match, so has worst case $O(2^n + m)$ complexity.
- So, why bother with DFA?
  - In many practical applications, the pattern is fixed and small, while the subject text is very large. So, the $O(mn)$ term is dominant over $O(2^n)$
  - For many important cases, DFAs are of polynomial size
  - In many applications, exponential blow-ups don't occur, e.g., compilers.

Derivative of Regular Expressions

The derivative of a regular expression R w.r.t. a symbol x, denoted $\partial_x[R]$ is another regular expression R' such that $\mathcal{L}(R) = \mathcal{L}(xR')$

Basically, $\partial_x[R]$ captures the suffixes of those strings that match R and start with x.

Examples

- $\partial_a[a(b|c)] = b|c$
- $\partial_a[(a|b)cd] = cd$
- $\partial_a[(a|b)*\ cd] = (a|b)*\ cd$
- $\partial_c[(\cancel{a|b})*\ cd] = d$
- $\partial_d[(\cancel{a|b})*\ \cancel{c}d] = \emptyset$

$\phi = \{\ \}$

$\epsilon = \{\epsilon\}$

Definition of RE Derivative (1)

inclEps(R): A predicate that returns true if $\epsilon \in \mathcal{L}(R)$

$$
\begin{aligned}
inclEps(a) &= false, \quad \forall a \in \Sigma \\
inclEps(R_1|R_2) &= inclEps(R_1) \vee inclEps(R_2) \\
inclEps(R_1R_2) &= inclEps(R_1) \wedge inclEps(R_2) \\
inclEps(R*) &= true
\end{aligned}
$$

$\mathcal{L}(a) = \{a\} \not\ni \epsilon$

$\mathcal{L}(R_1|R_2) = \boxed{\mathcal{L}(R_1)} \cup \boxed{\mathcal{L}(R_2)}$

$\mathcal{L}(R^*) = \{\epsilon\} \cup \cdots$

Note *inclEps* can be computed in linear-time.

Definition of RE Derivative (2)

$\partial_a[\epsilon] = \emptyset$

$$
\begin{array}{rcll}
\partial_a[a] & = & \epsilon & \\
\partial_a[b] & = & \emptyset & \\
\partial_a[R_1|R_2] & = & \partial_a[R_1]|\partial_a[R_2] & \\
\partial_a[R*] & = & \partial_a[R]R* & \\
\partial_a[R_1R_2] & = & \partial_a[R_1]R_2|\boxed{\partial_a[R_2]} & \text{if } inclEps(R_1) \\
 & = & \partial_a[R_1]R_2 & \text{otherwise}
\end{array}
$$

$\partial_a(ab|ac)$

$= \partial_a(ab)|\partial_a(ac)$

$= b|c$

$R^* = \epsilon|RR^*$

$\partial_a(\epsilon|RR^*) = \cancel{\emptyset}|\partial_a(RR^*)$

Note: $\boxed{\mathcal{L}(\epsilon) = \{\epsilon\}} \neq \boxed{\mathcal{L}(\emptyset) = \{\}}$

DFA Using Derivatives: Illustration

Consider $R_1 = (a|b) * a(a|b)$

$\partial_a[R_1] = R_1|(a|b) = R_2$

$\partial_b[R_1] = R_1$

$\partial_a[R_2] = R_1|(a|b)|\epsilon = R_3$

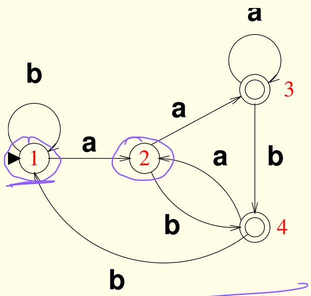
$\partial_b[R_2] = R_1|\epsilon = R_4$

$\partial_a[R_3] = R_1|(a|b)|\epsilon = R_3$

$\partial_b[R_3] = R_1|\epsilon = R_4$

$\partial_a[R_4] = R_1|(a|b) = R_2$

$\partial_b[R_4] = R_1$

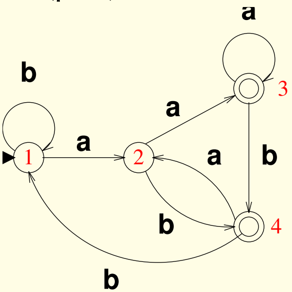


McNaughton-Yamada Construction

Can be viewed as a simpler way to represent derivatives

- Positions in RE are numbered, e.g., $^0(a^1|b^2)*a^3(a^4|b^5)\$^6$.
- A derivative is identified by its beginning position in the RE
  - Or more generally, a derivative is identified by a set of positions
- Each DFA state corresponds to a position set (pset)

$$
\begin{aligned}
R_1 &\equiv \{1,2,3\} \\
R_2 &\equiv \{1,2,3,4,5\} \\
R_3 &\equiv \{1,2,3,4,5,6\} \\
R_4 &\equiv \{1,2,3,6\}
\end{aligned}
$$



McNaughton-Yamada: Definitions

first(P): Yields the set of first symbols of RE denoted by pset P

Determines the transitions out of DFA state for P

Example: For the RE $(a^1|b^2)*\, a^3(a^4|b^5)\$^6, \quad \mathit{first}(\{1,2,3\}) = \{a, b\}$

$P|_s$: Subset of P that contain s, i.e., $\{p \in P \mid R \text{ contains } s \text{ at } p\}$

Example: $\{1,2,3\}|_a = \{1,3\}$, $\{1,2,4,5\}|_b = \{2,5\}$

follow(P): set of positions immediately after P, i.e., $\bigcup_{p\in P} \mathit{follow}(\{p\}))$

Definition is very similar to derivatives

Example: $\mathit{follow}(\{3,4\}) = \{4,5,6\}$

$\mathit{follow}(\{1\}) = \{1,2,3\}$

McNaughton-Yamada Construction (2)

BuildMY$(R, pset)$

Create an automaton state S labeled $pset$

Mark this state as final if \$ occurs in R at $pset$

foreach symbol $x \in \mathit{first}(pset) - \{\$\}$ **do**

  Call $\mathit{BuildMY}(R, \mathit{follow}(pset|_x))$ if hasn't previously been called

  Create a transition on x from S to

    the root of this subautomaton

DFA construction begins with the call $\mathit{BuildMY}(R, \mathit{follow}(\{0\}))$. The root of the resulting automaton is marked as a start state.

BuildMY Illustration on $R = {}^0(a^1|b^2)*\ a^3(a^4|b^5)\$^6$

Computations Needed

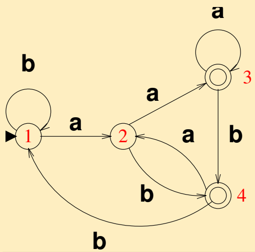
$follow(\{0\}) = \{1, 2, 3\}$
$follow(\{1\}) = follow(\{2\}) = \{1, 2, 3\}$
$follow(\{3\}) = \{4, 5\}$
$follow(\{4\}) = follow(\{5\}) = \{6\}$

$\{1, 2, 3\}|_a = \{1, 3\}, \quad \{1, 2, 3\}|_b = \{2\}$
$follow(\{1, 3\}) = \{1, 2, 3, 4, 5\}$

$\{1, 2, 3, 4, 5\}|_a = \{1, 3, 4\}$
$\{1, 2, 3, 4, 5\}|_b = \{2, 5\}$
$follow(\{1, 3, 4\}) = \{1, 2, 3, 4, 5, 6\}$
$follow(\{2, 5\}) = \{1, 2, 3, 6\}$

$\{1, 2, 3, 4, 5, 6\}|_a = \{1, 3, 4\}$
$\{1, 2, 3, 4, 5, 6\}|_b = \{2, 5\}$
$\{1, 2, 3, 6\}|_a = \{1, 3\} \quad \{1, 2, 3, 6\}|_b = \{2\}$

Resulting Automaton



State	Pset
1	{1,2,3}
2	{1,2,3,4,5}
3	{1,2,3,4,5,6}
4	{1,2,3,6}

McNaughton-Yamada (MY) Vs Derivatives

- Conceptually very similar
- MY takes a bit longer to describe, and its correctness a bit harder to follow.
- MY is also more mechanical, and hence is found in most implementations
- Derivatives approach is more general
  - Can support some extensions to REs, e.g., complement operator
  - Can avoid some redundant states during construction
    - Example: For $ac|bc$, DFA built by derivative approach has 3 states, but the one built by MY construction has 4 states

      The derivative approach merges the two c's in the RE, but with MY, the two c's have different positions, and hence operations on them are not shared.

Avoiding Redundant States

- Automata built by MY is not optimal
  - Automata minimization algorithms can be used to produce an optimal automaton.
- Derivatives approach associates DFA states with derivatives, but does not say how to determine equality among derivatives.
- There is a spectrum of techniques to determine RE equality
  - MY is the simplest: relies on syntactic identity
  - At the other end of the spectrum, we could use a complete decision procedure for RE equality.
    - In this case, the derivative approach yields the optimal RE!
  - In practice we would tend to use something in the middle
    - Trade off some power for ease/efficiency of implementation

RE to DFA conversion: Complexity

- Given DFA size can be exponential in the worst case, we obviously must accept worst-case exponential complexity.
- For the derivatives approach, it is not immediately obvious that it even terminates!
  - More obvious for McNaughton-Yamada approach, since DFA states correspond to position sets, of which there are only 2^n.
- Derivative computation is linear in RE size in the general case.
- So, overall complexity is $O(n2^n)$
- Complexity can be improved, but the worst-case 2^n takes away some of the rationale for doing so.
  - Instead, we focus on improving performance in many frequently occurring special cases where better complexity is achievable.

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}
```

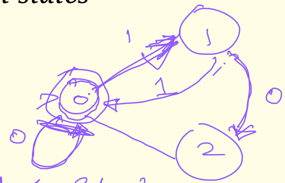
$k = 3n+1$

$2k+1 = 2(3n+1) + 1$

$= 6n+3$

$2k = 2(3n+1)$

$= 6n+2$



1101 ← $2k+1$

1100 ← $2k$

0

k

$2k+1$

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 */}
```

Implementing a Scanner

$$transition : state \times \Sigma \rightarrow state$$

```
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**.

```
/* */          /[*](.*)[*]/

               /[*]([^*/]*)[*]/

^*/            ([^*] | *[^/])*

[^*]   *[^/]   /* * * / * /
```

```
%x COM

/*        {BEGIN COM}

<COM>*/   {BEGIN 0}

<COM>.    {}
```