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:

<table>
<thead>
<tr>
<th>Token</th>
<th>Sample Lexemes</th>
<th>Pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>while</td>
<td>while</td>
<td>while</td>
</tr>
<tr>
<td>integer_constant</td>
<td>32894, −1093, 0</td>
<td>(−</td>
</tr>
<tr>
<td>identifier</td>
<td>buffer_size</td>
<td>[_a − zA − Z]+</td>
</tr>
</tbody>
</table>
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**.
Let $R$ be the set of all regular expressions over $\Sigma$. 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**: $r_1, r_2 \in R \Rightarrow (r_1 | r_2) \in R$
- **Kleene Closure**: $r \in R \Rightarrow r^* \in R$
**Semantics of Regular Expressions**

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

- $\mathcal{L}(\epsilon) = \{\epsilon\}$
- $\mathcal{L}(\alpha) = \{\alpha\}$ \hspace{1cm} ($\alpha \in \Sigma$)
- $\mathcal{L}(r_1 \mid r_2) = \mathcal{L}(r_1) \cup \mathcal{L}(r_2)$
- $\mathcal{L}(r_1 \cdot r_2) = \mathcal{L}(r_1) \cdot \mathcal{L}(r_2)$
- $\mathcal{L}(r^*) = \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*))$
Computing the Semantics

\[ L(a) \quad = \quad \{a\} \]

\[ L(a \mid b) \quad = \quad L(a) \cup L(b) \]

\[ = \quad \{a\} \cup \{b\} \]

\[ = \quad \{a, b\} \]
Computing the Semantics

\[ \mathcal{L}(a) = \{a\} \]

\[ \mathcal{L}(a \mid b) = \mathcal{L}(a) \cup \mathcal{L}(b) \]
\[ = \{a\} \cup \{b\} \]
\[ = \{a, b\} \]

\[ \mathcal{L}(ab) = \mathcal{L}(a) \cdot \mathcal{L}(b) \]
\[ = \{a\} \cdot \{b\} \]
\[ = \{ab\} \]
Computing the Semantics

\[
\mathcal{L}(a) = \{a\}
\]
\[
\mathcal{L}(a \mid b) = \mathcal{L}(a) \cup \mathcal{L}(b)
\]
\[
= \{a\} \cup \{b\}
\]
\[
= \{a, b\}
\]
\[
\mathcal{L}(ab) = \mathcal{L}(a) \cdot \mathcal{L}(b)
\]
\[
= \{a\} \cdot \{b\}
\]
\[
= \{ab\}
\]
\[
\mathcal{L}((a \mid b)(a \mid b)) = \mathcal{L}(a \mid b) \cdot \mathcal{L}(a \mid b)
\]
\[
= \{a, b\} \cdot \{a, b\}
\]
\[
= \{aa, ab, ba, bb\}
\]
Computing the Semantics of Closure

\[ L(r^*) = \{\epsilon\} \cup (L(r) \cdot L(r^*)) \]
Computing the Semantics of Closure

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

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

\[
L_0 = \{\epsilon\} \quad \text{Base case}
\]

\[
L_1 = \{\epsilon\} \cup (\{a, b\} \cdot L_0)
\]

\[
= \{\epsilon\} \cup (\{a, b\} \cdot \{\epsilon\})
\]

\[
= \{\epsilon, a, b\}
\]

\[
L_2 = \{\epsilon\} \cup (\{a, b\} \cdot L_1)
\]

\[
= \{\epsilon\} \cup (\{a, b\} \cdot \{\epsilon, a, b\})
\]

\[
= \{\epsilon, a, b, aa, ab, ba, bb\}
\]

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

$\mathcal{L}(a^*) = \{\epsilon, a, aa, \ldots\}$

$\mathcal{L}(b^*) = \{\epsilon, b, bb, \ldots\}$

$\mathcal{L}(a^*b^*) = \{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\}$

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

Union:

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

$\cup \{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, bbb, \ldots\}$

$\cup \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\}$

$\vdots$

$= \{\epsilon, a, b, aa, ab, ba, bb, \ldots\}$
Assign “names” to regular expressions.

For example,

\[
\text{digit} \longrightarrow 0 | 1 | \cdots | 9
\]
\[
\text{natural} \longrightarrow \text{digit digit}^*
\]

**SHORTHANDS:**

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

Example:

\[
\text{integer} \longrightarrow (\mathbf{+} | \mathbf{-})^? \text{digit}^+
\]
Regular Definitions: Examples

\[
\begin{align*}
\text{float} & \rightarrow \text{integer} \cdot \text{fraction} \\
\text{integer} & \rightarrow (Latin) \text{? no_leading_zero} \\
\text{no_leading_zero} & \rightarrow (\text{nonzero_digit digit}^*) | 0 \\
\text{fraction} & \rightarrow \text{no_trailing_zero exponent}^? \\
\text{no_trailing_zero} & \rightarrow (\text{digit}^* \text{nonzero_digit}) | 0 \\
\text{exponent} & \rightarrow (E | e) \text{integer} \\
\text{digit} & \rightarrow 0 | 1 | \cdots | 9 \\
\text{nonzero_digit} & \rightarrow 1 | 2 | \cdots | 9
\end{align*}
\]
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).

\[
\begin{align*}
[0-9]^+ & \quad \{ \text{emit(INTEGER_CONSTANT);} \} \\
[0-9]^+ \cdot [0-9]^+ & \quad \{ \text{emit(FLOAT_CONSTANT);} \}
\end{align*}
\]
Lex

Tool for building lexical analyzers.

Input: lexical specifications (.l file)

Output: C function (yy1lex) 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
```
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.: `| * ^`
<table>
<thead>
<tr>
<th>*  +  ?  (  )</th>
<th>Same as in regular expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>[    ]</td>
<td>Enclose ranges and exceptions</td>
</tr>
<tr>
<td>{    }</td>
<td>Enclose “names” of regular definitions</td>
</tr>
<tr>
<td>^</td>
<td>Used to negate a specified range (in Exception)</td>
</tr>
<tr>
<td>.</td>
<td>Match any single character except newline</td>
</tr>
<tr>
<td>\</td>
<td>Escape the next character</td>
</tr>
<tr>
<td>\n, \t</td>
<td>Newline and Tab</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>for</th>
<th>Sequence of f, o, r</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;</td>
<td></td>
</tr>
<tr>
<td>.*</td>
<td>Sequence of non-newline characters</td>
</tr>
<tr>
<td>[^*/]+</td>
<td>Sequence of characters except * and /</td>
</tr>
<tr>
<td>&quot;[^&quot;]*&quot;</td>
<td>Sequence of non-quote characters beginning and ending with a quote</td>
</tr>
<tr>
<td>({letter}</td>
<td>&quot;_&quot;)({letter}</td>
</tr>
</tbody>
</table>
A Complete Example

```c
{%
#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 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 `yy1val`
- 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?

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;if&quot;</td>
<td>{ return(TOKEN_IF); }</td>
</tr>
<tr>
<td>{letter}+</td>
<td>{ return(TOKEN_ID); }</td>
</tr>
<tr>
<td>&quot;while&quot;</td>
<td>{ return(TOKEN_WHILE); }</td>
</tr>
</tbody>
</table>

- A pattern that matches the longest string is chosen.
- Example: if's 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.
  - 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
  
  (f)lex

  *specifications*.l $\rightarrow$ lex.yy.c

- Generated scanner in lex.yy.c
  
  (g)cc

  lex.yy.c $\rightarrow$ executable

- `yywrap()` : hook for signalling end of file.

- Use `-lf1` (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 \((a \mid b)^*a(a \mid b)\).

\(L((a \mid b)^*a(a \mid b)) = \{aa, ab, aaa, aab, baa, bab, aaaa, aaab, abaa, abab, baaa, \ldots\}.\)
Finite State Automata: An Example

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

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

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

![Automaton Diagram]
(a | b)*a(a | 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 $\epsilon$.
  (Spontaneous transitions)

- All transition labels in a DFA belong to $\Sigma$.

- 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

- **$R = \text{Size of Regular Expression}$**
- **$N = \text{Length of Input String}$**

<table>
<thead>
<tr>
<th></th>
<th>NFA</th>
<th>DFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Automaton</td>
<td>$O(R)$</td>
<td>$O(2^R)$</td>
</tr>
<tr>
<td>Recognition time per input string</td>
<td>$O(N \times R)$</td>
<td>$O(N)$</td>
</tr>
</tbody>
</table>
**Thompson’s Construction:** For every regular expression $r$, derive an NFA $N(r)$ with unique start and final states.

- $\varepsilon$ (empty string)
- $\alpha \in \Sigma$ (any symbol in the alphabet)
- $(r_1 \mid r_2)$ (union of two regular expressions)
Regular Expressions to NFA (contd.)

\[ r_1 r_2 \]

\[ r^* \]

\[ N(r_1) \]

\[ N(r_2) \]

\[ \varepsilon \]

\[ \varepsilon \]
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

It can also be shown that every NFA can be converted into an equivalent RE

- Implication: REs are at least as expressive as NFAs

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

Input: \( a \ b \ a \ b \)
Path: 1 2 4 2 4 Accept
Recognition with an NFA

Is \( \text{abab} \in L((a | b)^*a(a | b)) \)?

Input: \( a \ b \ a \ b \)

Path 1: 1
Is \( abab \in \mathcal{L}((a \mid b)^*a(a \mid b)) \)?

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

Path 2: 1 1 1 Accept

Path 3: 1 2 3 \(\perp \perp\) Accept
Recognition with an NFA

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

\begin{align*}
\text{Input:} & \quad a \quad b \quad a \quad b \\
\text{Path 1:} & \quad 1 \quad 1 \quad 1 \quad 1
\end{align*}
Recognition with an NFA

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

Input: $a \ b \ a \ b$

Path 1: $1 \ 1 \ 1 \ 1 \ 1 \ 1$
Recognition with an NFA

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

Input: $a\ b\ a\ b$

Path 1: $1\ 1\ 1\ 1\ 1\ 1$

Path 2: $1\ 1\ 1$

Path 3: $1\ 2\ 3$
Recognition with an NFA

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

Input:

Path 1: 1 1 1 1 1 1
Path 2: 1 1 1 2
Path 3: 1 2 3

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

**Input:**

<table>
<thead>
<tr>
<th>Path 1:</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Path 2:</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>Accept</td>
</tr>
</tbody>
</table>
Recognition with an NFA

Is \( abab \in L((a \mid b)^*a(a \mid b)) \)?

| Path 1: | 1 | 1 | 1 | 1 | 1 |
| Path 2: | 1 | 1 | 1 | 2 | 3 | Accept
| Path 3: | 1 | 2 | 3 | \( \bot \) | \( \bot \) |
Recognition with an NFA

Is \( \text{abab} \in \mathcal{L}( (a | b)^*a(a | b) ) \)?

Input: \( \text{abab} \)

Path 1: \( 1 \ 1 \ 1 \ 1 \ 1 \ 1 \)

Path 2: \( 1 \ 1 \ 1 \ 2 \ 3 \) \( \text{Accept} \)

Path 3: \( 1 \ 2 \ 3 \ \bot \ \bot \)

All Paths: \( \{1\} \ \{1, 2\} \ \{1, 3\} \ \{1, 2\} \ \{1, 3\} \ \text{Accept} \)
Is $aaab \in \mathcal{L}((a \mid b)^*a(a \mid b))$?

Input:  
Path 1: 1 1 1 1 1 1 1  
Path 2: 1 1 1 1 1 2  
Path 3: 1 1 1 2 3 3 Accept  
Path 4: 1 1 2 3 \bot  
Path 5: 1 2 3 \bot \bot  

All Paths:  
$\{1\}$  
$\{1, 2\}$  
$\{1, 2, 3\}$  
$\{1, 2, 3\}$  
$\{1, 2, 3\}$ Accept
Is \( aabb \in L((a \mid b)^*a(a \mid b)) \)?

Input: \( a \ a \ a \ b \)

Path 1: 1 1 1 1 1 1 1
Path 2: 1 1 2 3 \( \bot \)
Path 3: 1 2 3 \( \bot \) \( \bot \)

All Paths: \( \{1\} \ \{1, 2\} \ \{1, 2, 3\} \ \{1, 3\} \ \{1\} \) REJECT
Converting NFA to DFA
Converting NFA to DFA (contd.)

Subset construction

Given a set $S$ of NFA states,

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

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

Each state in DFA corresponds to a *set of states* in NFA.
Start state of DFA = $\varepsilon$-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 $\alpha$ 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,

$\iff s$ is a final state of DFA
NFA → DFA: An Example

\[ \epsilon\text{-closure}\{1\} = \{1\} \]
\[ \text{goto}\{1\}, a\} = \{1, 2\} \]
\[ \text{goto}\{1\}, b\} = \{1\} \]
\[ \text{goto}\{1, 2\}, a\} = \{1, 2, 3\} \]
\[ \text{goto}\{1, 2\}, b\} = \{1, 3\} \]
\[ \text{goto}\{1, 2, 3\}, a\} = \{1, 2, 3\} \]
\[ \vdots \]
NFA → DFA: An Example (contd.)

\[ \epsilon\text{-closure}\{1\} = \{1\} \]
\[ \text{goto}\{1\}, a = \{1, 2\} \]
\[ \text{goto}\{1\}, b = \{1\} \]
\[ \text{goto}\{1, 2\}, a = \{1, 2, 3\} \]
\[ \text{goto}\{1, 2\}, b = \{1, 3\} \]
\[ \text{goto}\{1, 2, 3\}, a = \{1, 2, 3\} \]
\[ \text{goto}\{1, 2, 3\}, b = \{1\} \]
\[ \text{goto}\{1, 3\}, a = \{1, 2\} \]
\[ \text{goto}\{1, 3\}, b = \{1\} \]
NFA → DFA: An Example (contd.)

\[ \text{goto} (\{1\}, a) = \{1, 2\} \]
\[ \text{goto} (\{1\}, b) = \{1\} \]
\[ \text{goto} (\{1, 2\}, a) = \{1, 2, 3\} \]
\[ \text{goto} (\{1, 2\}, b) = \{1, 3\} \]
\[ \text{goto} (\{1, 2, 3\}, a) = \{1, 2, 3\} \]

...
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**

- $\partial_a[a(b|c)] = b|c$
- $\partial_a[(a|b)cd] = cd$
- $\partial_a[(a|b)^* cd] = (a|b)^* cd$
- $\partial_c[(a|b)^* cd] = d$
- $\partial_d[(a|b)^* cd] = \emptyset$
Definition of RE Derivative (1)

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

\[
\begin{align*}
\text{inclEps}(a) &= \text{false}, \quad \forall a \in \Sigma \\
\text{inclEps}(R_1 | R_2) &= \text{inclEps}(R_1) \lor \text{inclEps}(R_2) \\
\text{inclEps}(R_1 R_2) &= \text{inclEps}(R_1) \land \text{inclEps}(R_2) \\
\text{inclEps}(R^*) &= \text{true}
\end{align*}
\]

Note $\textit{inclEps}$ can be computed in linear-time.
Definition of RE Derivative (2)

\[
\begin{align*}
\partial_a[a] &= \epsilon \\
\partial_a[b] &= \emptyset \\
\partial_a[R_1|R_2] &= \partial_a[R_1]\partial_a[R_2] \\
\partial_a[R^*] &= \partial_a[R]R^* \\
\partial_a[R_1R_2] &= \partial_a[R_1]R_2|\partial_a[R_2] \quad \text{if } inclEps(R_1) \\
&= \partial_a[R_1]R_2 \quad \text{otherwise}
\end{align*}
\]

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

$$R_1 \equiv \{1, 2, 3\}$$
$$R_2 \equiv \{1, 2, 3, 4, 5\}$$
$$R_3 \equiv \{1, 2, 3, 4, 5, 6\}$$
$$R_4 \equiv \{1, 2, 3, 6\}$$
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)$\, \$, \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., \(\bigcup_{p \in P} \text{follow}(\{p\})\)

  Definition is very similar to derivatives

  **Example:** \(\text{follow}(\{3, 4\}) = \{4, 5, 6\}\)

  \(\text{follow}(\{1\}) = \{1, 2, 3\}\)
McNaughton-Yamada Construction (2)

BuildMY\((R, pset)\)

Create an automaton state \(S\) labeled \(pset\)
Mark this state as final if \(\$\) occurs in \(R\) at \(pset\)

\textbf{foreach} symbol \(x \in \text{first}(pset) - \{\$\} \) \textbf{do}

Call \(\text{BuildMY}(R, \text{follow}(pset|x))\) 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 \(\text{BuildMY}(R, \text{follow}(\{0\}))\). The root of the resulting automaton is marked as a start state.
BuildMY Illustration on \( R = 0(a^1|b^2) \ast a^3(a^4|b^5) \ast$ 6

**Computations Needed**

\[
\begin{align*}
\text{follow}([0]) &= \{1, 2, 3\} \\
\text{follow}([1]) &= \text{follow}([2]) = \{1, 2, 3\} \\
\text{follow}([3]) &= \{4, 5\} \\
\text{follow}([4]) &= \text{follow}([5]) = \{6\} \\
\{1, 2, 3\} |_a &= \{1, 3\}, \quad \{1, 2, 3\} |_b = \{2\} \\
\text{follow}([1, 3]) &= \{1, 2, 3, 4, 5\} \\
\{1, 2, 3, 4, 5\} |_a &= \{1, 3, 4\} \\
\{1, 2, 3, 4, 5\} |_b &= \{2, 5\} \\
\text{follow}([1, 3, 4]) &= \{1, 2, 3, 4, 5, 6\} \\
\text{follow}([2, 5]) &= \{1, 2, 3, 6\} \\
\{1, 2, 3, 4, 5, 6\} |_a &= \{1, 3, 4\} \\
\{1, 2, 3, 4, 5, 6\} |_b &= \{2, 5\} \\
\{1, 2, 3, 6\} |_a &= \{1, 3\} \quad \{1, 2, 3, 6\} |_b = \{2\}
\end{align*}
\]

**Resulting Automaton**

<table>
<thead>
<tr>
<th>State</th>
<th>Pset</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{1,2,3}</td>
</tr>
<tr>
<td>2</td>
<td>{1,2,3,4,5}</td>
</tr>
<tr>
<td>3</td>
<td>{1,2,3,4,5,6}</td>
</tr>
<tr>
<td>4</td>
<td>{1,2,3,6}</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccccc}
1 & a & 2 \\
2 & a & 3 \\
3 & b & 4 \\
4 & b & 5 \\
\end{array}
\]
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*

```plaintext
%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 */ }
```
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.