Lexical and Syntax Analysis
(of Programming Languages)

Lexical Analysis
Lexical and Syntax Analysis
(of Programming Languages)

Lexical Analysis
What is Parsing?

A parser also checks that the input string is well-formed, and if not, rejects it.
What is Parsing?

A parser also checks that the input string is well-formed, and if not, rejects it.
PARSING

= 

LEXICAL ANALYSIS

+ 

SYNTAX ANALYSIS
PARSING

= 

LEXICAL ANALYSIS

+ 

SYNTAX ANALYSIS
Lexical Analysis
(Also known as “scanning”)

- Identifies the **lexemes** in a sentence.
- **Lexeme**: a minimal meaningful unit of a language.
- Converts each lexeme to a **token**.
- Throws away ignorable text such as spaces, new-lines, and comments.
Lexical Analysis
(Also known as “scanning”)

- Identifies the **lexemes** in a sentence.

- **Lexeme**: a minimal meaningful unit of a language.

- Converts each lexeme to a **token**.

- Throws away ignorable text such as spaces, new-lines, and comments.
What is a token?

- Every token has an **identifier**, used to denote the **kind** of lexeme that it represents, e.g.

<table>
<thead>
<tr>
<th>Token identifier</th>
<th>denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>a + operator</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>a := operator</td>
</tr>
<tr>
<td>VAR</td>
<td>a variable</td>
</tr>
<tr>
<td>NUM</td>
<td>a number</td>
</tr>
</tbody>
</table>

- **Some** tokens have a **component value**, conventionally written in parenthesis after the identifier, e.g. `VAR(foo), NUM(12)`. 
What is a token?

- Every token has an **identifier**, used to denote the **kind** of lexeme that it represents, e.g.

<table>
<thead>
<tr>
<th>Token identifier</th>
<th>denotes</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLUS</td>
<td>a + operator</td>
</tr>
<tr>
<td>ASSIGN</td>
<td>a := operator</td>
</tr>
<tr>
<td>VAR</td>
<td>a variable</td>
</tr>
<tr>
<td>NUM</td>
<td>a number</td>
</tr>
</tbody>
</table>

- **Some** tokens have a **component value**, conventionally written in parenthesis after the identifier, e.g. **VAR(foo)**, **NUM(12)**.
Lexical Analysis

Example input:

\[\text{foo := 20 + bar}\]

Example output:

\[\text{VAR(foo), ASSIGN, NUM(20), PLUS, VAR(bar)}\]
Lexical Analysis

Stream of characters → Stream of tokens

Example input:

foo := 20 + bar

Example output:

VAR(foo), ASSIGN, NUM(20), PLUS, VAR(bar)
Lexical Analysis

Lexemes are specified by regular expressions. For example:

\[
\begin{align*}
\text{number} &= \text{digit} \cdot \text{digit}^* \\
\text{variable} &= \text{letter} \cdot (\text{letter} \mid \text{digit})^* \\
\text{digit} &= 0 \mid \ldots \mid 9 \\
\text{letter} &= a \mid \ldots \mid z
\end{align*}
\]

Example numbers:

1
4
43
634

Example variables:

x
foo
foo2
x1y20
Lexical Analysis

Lexemes are specified by regular expressions. For example:

\[
\begin{align*}
\text{number} & = \text{digit} \cdot \text{digit}^* \\
\text{variable} & = \text{letter} \cdot (\text{letter} \mid \text{digit})^* \\
\text{digit} & = 0 \mid \ldots \mid 9 \\
\text{letter} & = a \mid \ldots \mid z
\end{align*}
\]

Example numbers:
1
4
43
634

Example variables:
x
foo
foo2
x1y20
What exactly is a regular expression?
What exactly is a regular expression?
Notation

**Alphabet** $\Sigma$ is the set of all characters that can appear in an input string.

If a string $s$ **matches** a regular expressions $r$, we write $s \sim r$.

**Language** $L(r) = \{ s \mid s \sim r \}$, i.e. the set of all strings matching regular expression $r$.

We write $s_1s_2$ to denote the **concatenation** of strings $s_1$ and $s_2$. 
Notation

**Alphabet** $\Sigma$ is the set of all characters that can appear in an input string.

If a string $s$ **matches** a regular expressions $r$, we write $s \sim r$.

**Language** $L(r) = \{ s \mid s \sim r \}$, i.e. the set of all strings matching regular expression $r$.

We write $s_1 s_2$ to denote the **concatenation** of strings $s_1$ and $s_2$. 
The syntax of regular expressions is defined by the following grammar, where $x$ ranges over symbols in $\Sigma$.

\[
\begin{align*}
  r & \rightarrow \varepsilon \\
  r & \rightarrow x \\
  r & \rightarrow r \cdot r \\
  r & \rightarrow r / r \\
  r & \rightarrow r^* \\
  r & \rightarrow (r)
\end{align*}
\]
The syntax of regular expressions is defined by the following grammar, where $x$ ranges over symbols in $\Sigma$.

$$
\begin{align*}
  r & \rightarrow \varepsilon \\
  r & \rightarrow x \\
  r & \rightarrow r \cdot r \\
  r & \rightarrow r / r \\
  r & \rightarrow r^* \\
  r & \rightarrow (r)
\end{align*}
$$
## Intuitive definition

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Matching strings are</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$</td>
<td>The <strong>empty</strong> string $\epsilon$</td>
</tr>
<tr>
<td>$x$</td>
<td>The <strong>singleton</strong> string $x$ if $x \in \Sigma$</td>
</tr>
<tr>
<td>$r_1 / r_2$</td>
<td>Any string matching $r_1$ <strong>or</strong> $r_2$.</td>
</tr>
<tr>
<td>$r_1 \cdot r_2$</td>
<td>Any string that can be split into substrings $s_1$ and $s_2$ such that $s_1$ matches $r_1$ <strong>and</strong> $s_2$ matches $r_2$</td>
</tr>
<tr>
<td>$r^*$</td>
<td>The <strong>empty</strong> string <strong>or</strong> any string that can be split into substrings $s_1...s_n$ such that $s_i$ matches $r$ for all $i$ in $1...n$</td>
</tr>
</tbody>
</table>
## Intuitive definition

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>Matching strings are</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$</td>
<td>The <strong>empty</strong> string $\varepsilon$</td>
</tr>
<tr>
<td>$x$</td>
<td>The <strong>singleton</strong> string $x$ if $x \in \Sigma$</td>
</tr>
<tr>
<td>$r_1 / r_2$</td>
<td>Any string matching $r_1$ or $r_2$.</td>
</tr>
<tr>
<td>$r_1 \cdot r_2$</td>
<td>Any string that can be split into substrings $s_1$ and $s_2$ such that $s_1$ matches $r_1$ and $s_2$ matches $r_2$.</td>
</tr>
<tr>
<td>$r^*$</td>
<td>The <strong>empty</strong> string or any string that can be split into substrings $s_1 \ldots s_n$ such that $s_i$ matches $r$ for all $i$ in $1 \ldots n$.</td>
</tr>
</tbody>
</table>
Formal definition:

**Base cases**

\[ L(\varepsilon) = \{ \varepsilon \} \]

\[ L(x) = \{ x \} \]

where \( x \in \sum \)
Formal definition:

**Base cases**

\[ L(\varepsilon) = \{ \varepsilon \} \]

\[ L(x) = \{ x \} \]

*where* \( x \in \sum \)
Formal definition:

**Choice** and **Sequence**

\[ L(r_1 \mid r_2) = L(r_1) \cup L(r_2) \]

\[ L(r_1 \cdot r_2) = \{ s_1s_2 \mid s_1 \in L(r_1), s_2 \in L(r_2) \} \]
Formal definition: **Choice and Sequence**

\[ L(r_1 \mid r_2) = L(r_1) \cup L(r_2) \]

\[ L(r_1 \cdot r_2) = \{ s_1s_2 \mid s_1 \in L(r_1), s_2 \in L(r_2) \} \]
Formal definition: Kleene closure

\[ L(r^n) = \begin{cases} \{ \varepsilon \}, & \text{if } n = 0 \\ L(r \cdot r^{n-1}), & \text{if } n > 0 \end{cases} \]

\[ L(r^*) = \bigcup \{ L(r^n) \cdot n \in \{0, \ldots, \infty\} \} \]
Formal definition: Kleene closure

\[ L(r^n) = \begin{cases} \{ \varepsilon \}, & \text{if } n = 0 \\ L(r \cdot r^{n-1}), & \text{if } n > 0 \end{cases} \]

\[ L(r^*) = \bigcup \{ L(r^n) \mid n \in \{ 0 \ldots \infty \} \} \]
Example 1

Suppose $\sum = \{ a, b \}$.

<table>
<thead>
<tr>
<th>$r$</th>
<th>$L(r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a</td>
<td>b$</td>
</tr>
<tr>
<td>$(a</td>
<td>b) \cdot (a</td>
</tr>
<tr>
<td>$a^*$</td>
<td>${ \varepsilon, a, aa, aaa, ... }$</td>
</tr>
<tr>
<td>$(a \cdot b)^*$</td>
<td>${ \varepsilon, ab, abab, ... }$</td>
</tr>
<tr>
<td>$(a</td>
<td>b)^*$</td>
</tr>
</tbody>
</table>
Example 1

Suppose $\sum = \{ a, b \}$.

<table>
<thead>
<tr>
<th>$r$</th>
<th>$L(r)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a/b$</td>
<td>${ a, b }$</td>
</tr>
<tr>
<td>$(a/b) \cdot (a/b)$</td>
<td>${ aa, ab, ba, bb }$</td>
</tr>
<tr>
<td>$a^*$</td>
<td>${ \varepsilon, a, aa, aaa, \ldots }$</td>
</tr>
<tr>
<td>$(a \cdot b)^*$</td>
<td>${ \varepsilon, ab, abab, \ldots }$</td>
</tr>
<tr>
<td>$(a/b)^*$</td>
<td>${ \varepsilon, a, b, ab, ba, \ldots }$</td>
</tr>
</tbody>
</table>
Example 2

Example of a language that cannot be defined by a regular expression:

\[ \{ a^n b^n \; \cdot \; n \in \mathbb{N} \} \]

The set of strings containing \( n \) consecutive \( a \) symbols followed by \( n \) consecutive \( b \) symbols, for all \( n \).
Example 2

Example of a language that cannot be defined by a regular expression:

\[ \{ a^n b^n \; \bullet \; n \in \mathbb{N} \} \]

The set of strings containing \( n \) consecutive \( a \) symbols followed by \( n \) consecutive \( b \) symbols, for all \( n \).
Exercise 1

**Characterise** the languages defined by the following regular expressions

- $a \cdot (a/b)^* \cdot a$
- $a^* \cdot b \cdot a^* \cdot b \cdot a^* \cdot b \cdot a^*$
- $((\varepsilon/a) \cdot b^*)^*$
Exercise 1

**Characterise** the languages defined by the following regular expressions

- $a \cdot (a|b)^* \cdot a$
- $a^* \cdot b \cdot a^* \cdot b \cdot a^* \cdot b \cdot a^*$
- $((\varepsilon|a) \cdot b^*)^*$
Proof rules:
Base cases

The empty string $\varepsilon$ matches $\varepsilon$.

$$\varepsilon \sim \varepsilon \quad \text{[Empty]}$$

If $x \in \Sigma$ then $x$ matches $x$.

$$x \in \Sigma \quad \frac{\text{[Single]}}{x \sim x}$$
Proof rules:
Base cases

The empty string $\varepsilon$ matches $\varepsilon$.

$$\varepsilon \sim \varepsilon$$  \hspace{1cm} \text{[Empty]}

If $x \in \Sigma$ then $x$ matches $x$.

$$
\begin{align*}
x & \in \Sigma \\
\hline
x & \sim x
\end{align*}
$$  \hspace{1cm} \text{[Single]}
Proof rules:

Sequence

If $s_1$ matches $r_1$ and $s_2$ matches $r_2$
then $s_1s_2$ matches $r_1 \cdot r_2$.

\[
\frac{s_1 \sim r_1 \quad s_2 \sim r_2}{s_1s_2 \sim r_1 \cdot r_2} \quad [\text{Seq}]
\]
Proof rules: 
**Sequence**

If $s_1$ matches $r_1$ and $s_2$ matches $r_2$ then $s_1s_2$ matches $r_1 \cdot r_2$.

$$
\frac{s_1 \sim r_1 \quad s_2 \sim r_2}{s_1s_2 \sim r_1 \cdot r_2} \quad \text{[Seq]}
$$
Proof rules: Choice

If $s$ matches $r_1$
then $s$ matches $r_1 \mid r_2$

\[ s \sim r_1 \]

\[ s \sim r_1 \mid r_2 \quad [\text{Or}_1] \]

If $s$ matches $r_2$
then $s$ matches $r_1 \mid r_2$

\[ s \sim r_2 \]

\[ s \sim r_1 \mid r_2 \quad [\text{Or}_2] \]
Proof rules: Choice

**If** $s$ matches $r_1$

**then** $s$ matches $r_1 \mid r_2$

\[
\frac{s \sim r_1}{s \sim r_1 \mid r_2} \quad [\text{Or}_1]
\]

**If** $s$ matches $r_2$

**then** $s$ matches $r_1 \mid r_2$

\[
\frac{s \sim r_2}{s \sim r_1 \mid r_2} \quad [\text{Or}_2]
\]
Proof rules: Kleene closure

If $s$ matches $\varepsilon$

then $s$ matches $r^*$.  

\[
\begin{align*}
  s & \sim \varepsilon \\
  \hline
  s & \sim r^* \\
\end{align*}
\]

[Kleene$_1$]

If $s$ matches $r \cdot r^*$

then $s$ matches $r^*$.  

\[
\begin{align*}
  s & \sim r \cdot r^* \\
  \hline
  s & \sim r^* \\
\end{align*}
\]

[Kleene$_2$]
Proof rules: Kleene closure

If $s$ matches $\varepsilon$  
then $s$ matches $r^*$.  

\[
\frac{s \sim \varepsilon}{s \sim r^*} \quad \text{[Kleene}_1]\]

If $s$ matches $r \cdot r^*$  
then $s$ matches $r^*$.  

\[
\frac{s \sim r \cdot r^*}{s \sim r^*} \quad \text{[Kleene}_2]\]
Exercise 2

**Proove** that the string

\[ cab \]

matches the regular expression

\[ ((a \cdot b)/c)^* \]
Exercise 2

Proove that the string

$\text{cab}$

matches the regular expression

$((a \cdot b)/c)^*$
Exercise 2

$cab \sim ((a \cdot b) \| c)^*$

$\iff \{ \text{Kleene}_2 \}$

$cab \sim ((a \cdot b) \| c) \cdot ((a \cdot b) \| c)^*$

$\iff \{ \text{Seq} \}$

$c \sim (a \cdot b) \| c, \ ab \sim ((a \cdot b) \| c)^*$

$\iff \{ \text{Or}_2 \}$

$c \sim c, \ ab \sim ((a \cdot b) \| c)^*$

$\iff \{ \text{Single} \}$

$ab \sim ((a \cdot b) \| c)^*$

Continued...
Exercise 2

\[ cab \sim ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Kleene}_2 \} \]
\[ cab \sim ((a \cdot b) | c) \cdot ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Seq} \} \]
\[ c \sim (a \cdot b) | c, \ ab \sim ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Or}_2 \} \]
\[ c \sim c, \ ab \sim ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Single} \} \]
\[ ab \sim ((a \cdot b) | c)^* \]

Continued...
Exercise 2

\[ ab \sim ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Kleene}_2 \} \]
\[ ab \sim ((a \cdot b) | c) \cdot ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Seq} \} \]
\[ ab \sim (a \cdot b) | c, \ \varepsilon \sim ((a \cdot b) | c)^* \]
\[ \Leftarrow \{ \text{Or}_1, \text{Kleene}_1 \} \]
\[ ab \sim (a \cdot b) \]
\[ \Leftarrow \{ \text{Seq} \} \]
\[ a \sim a, b \sim b \]
\[ \Leftarrow \{ \text{Single, Single} \} \]
\[ \text{true} \]
Exercise 2

\[ ab \sim ((a \cdot b) | c)^* \]

\[ \iff \{ Kleene_2 \} \]

\[ ab \sim ((a \cdot b) | c) \cdot ((a \cdot b) | c)^* \]

\[ \iff \{ Seq \} \]

\[ ab \sim (a \cdot b) | c, \ \varepsilon \sim ((a \cdot b) | c)^* \]

\[ \iff \{ Or_1, Kleene_1 \} \]

\[ ab \sim (a \cdot b) \]

\[ \iff \{ Seq \} \]

\[ a \sim a, b \sim b \]

\[ \iff \{ Single, Single \} \]

\textit{true}
Sound and Complete

The proof rules are:

**Sound**: if we can prove $s \sim r$ using the rules then $s \in L(r)$.

**Complete**: if $s \in L(r)$ then we can prove $s \sim r$ using the rules.
Sound and Complete

The proof rules are:

**Sound**: if we can prove \( s \sim r \) using the rules then \( s \in L(r) \).

**Complete**: if \( s \in L(r) \) then we can prove \( s \sim r \) using the rules.
If we define our proof rules in Prolog then we get a regular expression implementation.

\[
\begin{align*}
[] & \sim []. \\
[X] & \sim X. \\
S & \sim R1!R2 :\sim S \sim R1. \\
S & \sim R1!R2 :\sim S \sim R2. \\
S & \sim R1.R2 :\sim \text{append}(S1, S2, S), \\
    & \quad S1 \sim R1, S2 \sim R2. \\
[] & \sim R^*. \\
S & \sim R^* :\sim \text{append}(S1, S2, S), S1=[X|Xs], \\
    & \quad S1 \sim R, S2 \sim R^*. \\
\end{align*}
\]

**NOTES:**
- Operator ! used to represent vertical bar.
- Read “:-” as “if”.
- Strings represented by lists of symbols.
- Termination ensured by requiring S1 to be non-empty in final clause.
If we define our proof rules in **Prolog** then we get a regular expression implementation.

\[
\begin{align*}
[] & \sim []. \\
[X] & \sim X. \\
S & \sim R_1!R_2 : - S \sim R_1. \\
S & \sim R_1!R_2 : - S \sim R_2. \\
S & \sim R_1.R_2 : - \text{append}(S_1, S_2, S), \\
& \quad S_1 \sim R_1, S_2 \sim R_2. \\
[] & \sim R^*. \\
S & \sim R^* : - \text{append}(S_1, S_2, S), S_1=[X|Xs], \\
& \quad S_1 \sim R, S_2 \sim R^*. 
\end{align*}
\]

**NOTES:**

- Operator ! used to represent vertical bar.
- Read “:-” as “if”.
- Strings represented by lists of symbols.
- Termination ensured by requiring S1 to be non-empty in final clause.
Prolog

Sadly the Prolog implementation is not very efficient:

- when applying the proof rules by hand we used human intuition to know **where to split the string**;
- Prolog does not have this intuition;
- instead, Prolog **guesses**, trying all possible ways to split a string, and **backtracks** on failure.
Prolog

Sadly the Prolog implementation is not very efficient:

- when applying the proof rules by hand we used human intuition to know where to split the string;
- Prolog does not have this intuition;
- instead, Prolog guesses, trying all possible ways to split a string, and backtracks on failure.
# Common extensions

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>is the same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r^+$</td>
<td>$r \cdot r^*$</td>
</tr>
<tr>
<td>$r^?$</td>
<td>$\varepsilon/r$</td>
</tr>
<tr>
<td>$[c_1c_2c_3-c_n]$</td>
<td>$c_1/c_2/c_3/.../c_n$</td>
</tr>
<tr>
<td>$[^c_1c_2]$</td>
<td>${x \cdot x \in \sum, x \notin {c_1, c_2}}$</td>
</tr>
</tbody>
</table>
Common extensions

<table>
<thead>
<tr>
<th>Regular expression</th>
<th>is the same as</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r^+ )</td>
<td>( r \cdot r^* )</td>
</tr>
<tr>
<td>( r^? )</td>
<td>( \varepsilon / r )</td>
</tr>
<tr>
<td>( [c_1 \ldots c_n] )</td>
<td>( c_1 / c_2 / c_3 / \ldots / c_n )</td>
</tr>
<tr>
<td>( [^c_1 c_2] )</td>
<td>( { x \mid x \in \sum, x \not\in {c_1, c_2} } )</td>
</tr>
</tbody>
</table>
Escaping

What if $\Sigma$ contains regular expression symbols such as $/ \ast \cdot ( + [ ?$

We can escape such symbols by prefixing with a backslash:

$\backslash / \backslash \ast \backslash \cdot \backslash ( \backslash [ \backslash ?$

And if we want $\backslash$ then write $\backslash \backslash$.

**Example:** $\backslash [ \ast \cdot \backslash ] \backslash \ast$ means zero or left brackets followed by zero or more right brackets.
Escaping

What if $\Sigma$ contains regular expression symbols such as

$$/ \ast \cdot ( + [ ?$$

We can escape such symbols by prefixing with a backslash:

$$\\backslash / \backslash* \backslash\cdot \backslash( \backslash[ \backslash?$$

And if we want $\backslash$ then write $\backslash\backslash$.

Example: $\backslash[\ast \cdot \backslash]*$ means zero or left brackets followed by zero or more right brackets.
Regular definitions

For convenience, we may wish to name a regular expression so that we can refer to it many times:

\[ \text{name} = r \]

We write \text{name} but sometimes the notation \{name\} is used (e.g. in Flex). Example:

\[
\begin{align*}
\text{number} & = \text{digit} \cdot \text{digit}^* \\
\text{digit} & = 0 \mid \ldots \mid 9
\end{align*}
\]
Regular definitions

For convenience, we may wish to name a regular expression so that we can refer to it many times:

\[\text{name} = r\]

We write \text{name} but sometimes the notation \{name\} is used (e.g. in Flex). Example:

\[
\begin{align*}
\text{number} & = \text{digit} \cdot \text{digit}^* \\
\text{digit} & = 0 \mid \ldots \mid 9 
\end{align*}
\]
Implementing regular expressions

How do we convert a regular expression $r$ into an efficient program that prints YES when applied to any string in $L(r)$ and NO in all other cases?

Two options:

- **By hand** (LSA Lab 1)
- **Automatically** (Chapters 3 & 4 of lecture notes, and LSA Lab 2)
Implementing regular expressions

How do we convert a regular expression $r$ into an efficient program that prints YES when applied to any string in $L(r)$ and NO in all other cases?

Two options:

- **By hand** (LSA Lab 1)
- **Automatically** (Chapters 3 & 4 of lecture notes, and LSA Lab 2)
Outline

Automatic conversion of regular expressions to efficient string-matching functions:

- **Step 1:** RE $\rightarrow$ NFA
- **Step 2:** NFA $\rightarrow$ DFA
- **Step 3:** DFA $\rightarrow$ C Function

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>Regular Expression</td>
</tr>
<tr>
<td>NFA</td>
<td>Non-deterministic Finite Automaton</td>
</tr>
<tr>
<td>DFA</td>
<td>Deterministic Finite Automaton</td>
</tr>
</tbody>
</table>
Automatic conversion of regular expressions to efficient string-matching functions:

- **Step 1:** RE $\rightarrow$ NFA
- **Step 2:** NFA $\rightarrow$ DFA
- **Step 3:** DFA $\rightarrow$ C Function

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RE</td>
<td>Regular Expression</td>
</tr>
<tr>
<td>NFA</td>
<td>Non-deterministic Finite Automaton</td>
</tr>
<tr>
<td>DFA</td>
<td>Deterministic Finite Automaton</td>
</tr>
</tbody>
</table>
STEP 1: RE $\rightarrow$ NFA

Thompson’s construction
STEP 1: RE $\rightarrow$ NFA

Thompson’s construction
What is an NFA?

A directed graph with nodes denoting states

A state \( s \)

The start state \( s \)

An accepting state \( s \)

The start state \( s \) that is also an accepting state

and edges labelled with a symbol \( x \in \Sigma \cup \{\varepsilon\} \) denoting transitions

\[ s_1 \xrightarrow{x} s_2 \]
What is an NFA?

A directed graph with nodes denoting states

A state $s$

The start state $s$

An accepting state $s$

The start state $s$ that is also an accepting state

and edges labelled with a symbol $x \in \Sigma \cup \{\varepsilon\}$ denoting transitions

$S_1 \xrightarrow{x} S_2$
Meaning of an NFA

A string $x_1x_2...x_n$ is accepted by an NFA if there is a path labelled $x_1,x_2,...,x_n$ (including any number of $\varepsilon$ transitions) from the start state to an accepting state.
A string $x_1x_2...x_n$ is accepted by an NFA if there is a path labelled $x_1,x_2,...,x_n$ (including any number of $\varepsilon$ transitions) from the start state to an accepting state.
Example of an NFA

The following NFA accepts exactly the strings that match the regular expression $a \cdot a^* \mid b \cdot b^*$. 

![NFA Diagram](image-url)
Example of an NFA

The following NFA accepts exactly the strings that match the regular expression $a \cdot a^* \mid b \cdot b^*$.
Thompson’s construction: Notation

Let $N(r)$ be the NFA accepting exactly the set of strings in $L(r)$.

We abstractly represent an NFA $N(r)$ with start state $s_0$ and final state $s_a$ by the diagram:
Thompson’s construction: Notation

Let $N(r)$ be the NFA accepting exactly the set of strings in $L(r)$.

We abstractly represent an NFA $N(r)$ with start state $s_0$ and final state $s_\alpha$ by the diagram:
Thompson’s construction:
Base cases

\[ N(\varepsilon) \]

\[ \varepsilon \]

\[ N(x) \]

\[ x \in \Sigma \]
Thompson’s construction: Base cases

\[ N(\varepsilon) = s_0 \xrightarrow{\varepsilon} s_a \]

\[ N(x) = s_0 \xrightarrow{x} s_a \]

where \( x \in \Sigma \)
Thompson’s construction: Choice

\[ N(r|t) \]

\[ s_0 \rightarrow N(r) \rightarrow s_a \]

\[ \epsilon \rightarrow N(t) \rightarrow \epsilon \]
Thompson’s construction: Choice

\[ N(r|t) \]

\[ s_0 \quad \text{\(N(r|t)\)} \quad s_a \]

\[ = \]

\[ N(r) \]

\[ s_0 \quad \varepsilon \quad N(r) \quad \varepsilon \quad s_a \]

\[ N(t) \]

\[ s_0 \quad \varepsilon \quad N(t) \quad \varepsilon \quad s_a \]
Thompson’s construction: Sequence

\[ s_0 \xrightarrow{N(r \cdot t)} s_a = s_0 \xrightarrow{N(r)} N(t) \xrightarrow{} s_a \]
Thompson’s construction: 
Sequence

\[ s_0 N(r \cdot t) s_0 = s_a N(r) s_a = s_a N(t) s_a \]
Thompson’s construction: Kleene closure

\[ N(r^*) \]

\[ \varepsilon \]

\[ s_0 \rightarrow N(r) \rightarrow s_a \]
Thompson’s construction: Kleene closure

\[ N(r^*) = \]

\[
\begin{array}{c}
S_0 \\
N(r) \\
S_a
\end{array}
\]
Exercise 3

Apply Thompson’s construction to the following regular expression.

\(((a \cdot b) | c)\)^*
Exercise 3

Apply Thompson’s construction to the following regular expression.

\[((a \cdot b) | c)\]
Problem with NFAs

It is not straightforward to turn an NFA into a deterministic program because:

- There may be many possible next-states for a given input.
- Which one do we choose?
- Try them all?

**Idea:** convert an NFA into a DFA: a DFA can be easily converted into an efficient executable program.
Problem with NFAs

It is not straightforward to turn an NFA into a deterministic program because:

- There may be many possible next-states for a given input.
- Which one do we choose?
- Try them all?

**Idea:** convert an NFA into a DFA: a DFA can be easily converted into an efficient executable program.
STEP 2: NFA $\rightarrow$ DFA

The subset construction.
STEP 2: NFA $\rightarrow$ DFA

The subset construction.
What is a DFA?

A deterministic finite automaton (DFA) is an NFA in which

- there are no $\varepsilon$ transitions, and
- for each state $s$ and input symbol $a$ there is at most one transition out of $s$ labelled $a$. 
What is a DFA?

A deterministic finite automaton (DFA) is an NFA in which

- there are no $\varepsilon$ transitions, and
- for each state $s$ and input symbol $a$ there is at most one transition out of $s$ labelled $a$. 

Example of a DFA

The following DFA accepts exactly the strings that match the regular expression $a \cdot a^* \mid b \cdot b^*$. 
Example of a DFA

The following DFA accepts exactly the strings that match the regular expression $a \cdot a^* \mid b \cdot b^*$. 
NFA → DFA: key observation

After consuming an input string, an NFA can be in one of a set of states. Example 3:

<table>
<thead>
<tr>
<th>Input</th>
<th>States</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>0, 1, 2</td>
</tr>
<tr>
<td>aba</td>
<td></td>
</tr>
<tr>
<td>aab</td>
<td></td>
</tr>
<tr>
<td>aaba</td>
<td></td>
</tr>
<tr>
<td>ε</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- Initial state: 0
- States: 0, 1, 2, 3
- Transitions:
  - From 0 on a: 1, 2
  - From 0 on b: 1
  - From 1 on a: 2
  - From 1 on b: 3
  - From 2 on a: 3
  - From 2 on b: 3
  - From 3 on a: 0
  - From 3 on b: 0

Graph:

```
0 --a--> 1
 |       |       |
 v       v       v
2 --b--> 3
```
After consuming an input string, an NFA can be in one of a set of states. Example 3:
Idea: construct a DFA in which each state corresponds to a set of NFA states.

After consuming $a_1 \cdots a_n$ the DFA is in a state which corresponds to the set of states that the NFA can reach on input $a_1 \cdots a_n$. 
**NFA → DFA: key idea**

**Idea:** construct a DFA in which each state corresponds to a set of NFA states.

After consuming $a_1 \cdots a_n$ the DFA is in a state which corresponds to the set of states that the NFA can reach on input $a_1 \cdots a_n$. 
Example 3, revisited

Create a DFA state corresponding to each set of NFA states.

<table>
<thead>
<tr>
<th>Input</th>
<th>NFA States</th>
<th>DFA State</th>
</tr>
</thead>
<tbody>
<tr>
<td>aa</td>
<td>0, 1, 2</td>
<td>A</td>
</tr>
<tr>
<td>aba</td>
<td>0,1</td>
<td>B</td>
</tr>
<tr>
<td>aab</td>
<td>0,3</td>
<td>C</td>
</tr>
<tr>
<td>aaba</td>
<td>0,1</td>
<td>B</td>
</tr>
<tr>
<td>ε</td>
<td>0</td>
<td>D</td>
</tr>
</tbody>
</table>

**Question:** which states would be initial and final DFA states?
Example 3, revisited

Create a DFA state corresponding to each set of NFA states.

<table>
<thead>
<tr>
<th>Input</th>
<th>NFA States</th>
<th>DFA State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$aa$</td>
<td>0, 1, 2</td>
<td>A</td>
</tr>
<tr>
<td>$aba$</td>
<td>0,1</td>
<td>B</td>
</tr>
<tr>
<td>$aab$</td>
<td>0,3</td>
<td>C</td>
</tr>
<tr>
<td>$aab!!!!!!!!a$</td>
<td>0,1</td>
<td>B</td>
</tr>
<tr>
<td>$\varepsilon$</td>
<td>0</td>
<td>D</td>
</tr>
</tbody>
</table>

**Question:** which states would be initial and final DFA states?
## Notation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$-closure($s$)</td>
<td>Set of NFA states reachable from NFA state $s$ on zero or more $\varepsilon$-transitions.</td>
</tr>
<tr>
<td>$\varepsilon$-closure($T$)</td>
<td>$\bigcup_{s \in T} \varepsilon$-closure($s$)</td>
</tr>
<tr>
<td>$move(T, a)$</td>
<td>Set of NFA states to which there is a transition on symbol $a$ from some state $s$ in $T$.</td>
</tr>
</tbody>
</table>
## Notation

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon$-closure(s)</td>
<td>Set of NFA states reachable from NFA state $s$ on zero or more $\varepsilon$-transitions.</td>
</tr>
<tr>
<td>$\varepsilon$-closure(T)</td>
<td>$\bigcup_{s \in T} \varepsilon$-closure(s)</td>
</tr>
<tr>
<td>move(T, a)</td>
<td>Set of NFA states to which there is a transition on symbol $a$ from some state $s$ in $T$.</td>
</tr>
</tbody>
</table>
Exercise 4

Consider the following NFA.

Compute:

- $\varepsilon$-closure(0)
- $\varepsilon$-closure({1, 2})
- move({0,3}, a)
- $\varepsilon$-closure(move({0,3}, a))
Exercise 4

Consider the following NFA.

Compute:

- $\varepsilon$-closure($0$)
- $\varepsilon$-closure($\{1, 2\}$)
- move($\{0,3\}, a$)
- $\varepsilon$-closure(move($\{0,3\}, a$))
Subset construction: input and output

**Input**: an NFA $N$.

**Output**: a DFA $D$ accepting the same language as $N$. Specifically, the set of states of $D$, termed $D_{states}$, and its transition function $D_{tran}$ that maps any state-symbol pair to a next state.
Subset construction: input and output

**Input:** an NFA $N$.

**Output:** a DFA $D$ accepting the same language as $N$. Specifically, the set of states of $D$, termed $D_{states}$, and its transition function $D_{tran}$ that maps any state-symbol pair to a next state.
Subset construction: input and output

- Each state in $D$ is denoted by a subset of $N$'s states.
- To ensure termination, every state is either marked or unmarked.
- Initially, $D_{states}$ contains a single unmarked start state $\varepsilon$-closure($s_0$) where $s_0$ is the start state of $N$.
- The accepting states of $D$ are the states that contain at least one accepting state of $N$. 
Subset construction: input and output

- Each state in $D$ is denoted by a subset of $N$’s states.
- To ensure termination, every state is either marked or unmarked.
- Initially, $D_{states}$ contains a single unmarked start state $\varepsilon$-closure($s_0$) where $s_0$ is the start state of $N$.
- The accepting states of $D$ are the states that contain at least one accepting state of $N$. 
Subset construction: algorithm

while (there is an unmarked state $T$ in $D_{states}$) {
    mark $T$;
    for (each input symbol $a$) {
        $U = \varepsilon$-closure($move(T, a)$);
        $D_{tran}[T, a] = U$
        if ($U$ is not in $D_{states}$)
            add $U$ as unmarked state to $D_{states}$;
    }
}
Subset construction: 
algorithm

while (there is an unmarked state $T$ in $D_{\text{states}}$) {
    mark $T$;
    for (each input symbol $a$) {
        $U = \varepsilon$-closure(move($T, a$));
        $D_{\text{tran}}[T, a] = U$
        if ($U$ is not in $D_{\text{states}}$)
            add $U$ as unmarked state to $D_{\text{states}}$;
    }
}
Exercise 5

Convert the following NFA into a DFA by applying the subset construction algorithm.
Exercise 5

Convert the following NFA into a DFA by applying the subset construction algorithm.
Exercise 6

It is **not obvious** how to simulate an NFA in **linear time** with respect to the length of the input string.

But it may be converted to a DFA that can be simulated **easily** in linear time.

**What’s the catch?** Can you think of any problems with the DFA produced by subset construction?
Exercise 6

It is **not obvious** how to simulate an NFA in **linear time** with respect to the length of the input string.

But it may be converted to a DFA that can be simulated **easily** in linear time.

**What’s the catch?** Can you think of any problems with the DFA produced by subset construction?
Caveats

- Number of DFA states **could be exponential** in number of NFA states!

- DFA produced is **not minimal** in number of states. (Can apply a minimisation algorithm.)

- Often no problem in practice.
Caveats

- Number of DFA states could be exponential in number of NFA states!

- DFA produced is not minimal in number of states. (Can apply a minimisation algorithm.)

- Often no problem in practice.
Homework Exercise

Convert the following NFA into a DFA by applying the subset construction algorithm.
Homework Exercise

Convert the following NFA into a DFA by applying the subset construction algorithm.
STEP 3: DFA $\rightarrow$ C CODE
STEP 3: DFA → C CODE
Exercise 7

Implement the DFA

```
int match(char *next) {
    ...
}
```

returning 1 if the string pointed to by next is accepted by the DFA and 0 otherwise.
Exercise 7

Implement the DFA as a C function

```c
int match(char *next) {
    ...
}
```

returning 1 if the string pointed to by `next` is accepted by the DFA and 0 otherwise.
int match(char* next)
{
    goto A;    /* start state */

    A:    if (*next == '\0') return 1;
          if (*next == 'a')  { next++; goto B; }
          if (*next == 'c')  { next++; goto C; }
          return 0;

    B:    if (*next == '\0') return 0;
          if (*next == 'b')  { next++; goto D; }
          return 0;

    C:    if (*next == '\0') return 1;
          if (*next == 'a')  { next++; goto B; }
          if (*next == 'c')  { next++; goto D; }
          return 0;

    D:    if (*next == '\0') return 1;
          if (*next == 'a')  { next++; goto B; }
          if (*next == 'c')  { next++; goto D; }
          return 0;
}
int match(char* next)
{
    goto A;  /* start state */

    A:  if (*next == '\0') return 1;
        if (*next == 'a') { next++; goto B; }
        if (*next == 'c') { next++; goto C; }
        return 0;

    B:  if (*next == '\0') return 0;
        if (*next == 'b') { next++; goto D; }
        return 0;

    C:  if (*next == '\0') return 1;
        if (*next == 'a') { next++; goto B; }
        if (*next == 'c') { next++; goto D; }
        return 0;

    D:  if (*next == '\0') return 1;
        if (*next == 'a') { next++; goto B; }
        if (*next == 'c') { next++; goto D; }
        return 0;
}
SUMMARY
SUMMARY
Summary

- In lexical analysis, the lexemes of the language are identified and converted into tokens.
- Lexemes are typically specified by regular expressions.
- Matching of regular expressions formalised by proof rules.
- Defining proof rules in Prolog gives a simple but inefficient implementation.
Summary

- In lexical analysis, the lexemes of the language are identified and converted into tokens.
- Lexemes are typically specified by regular expressions.
- Matching of regular expressions formalised by proof rules.
- Defining proof rules in Prolog gives a simple but inefficient implementation.
Summary

- Automatically converting regular expressions into efficient C code involves three main steps:

1. **RE → NFA**
   (Thompson’s Construction)

2. **NFA → DFA**
   (Subset Construction)

3. **DFA → C Function**
Summary

- Automatically converting regular expressions into efficient C code involves three main steps:

1. **RE → NFA**
   (Thompson’s Construction)

2. **NFA → DFA**
   (Subset Construction)

3. **DFA → C Function**
In the next lecture, we will learn how to use a tool called **Flex** that puts the regular expression theory into practice.