2. Lexical Analysis

Eva Rose  Kristoffer Rose

NYU Courant Institute
Compiler Construction (CSCI-GA.2130-001)
http://cs.nyu.edu/courses/fall14/CSCI-GA.2130-001/lecture-2.pdf

September 11, 2014
1. Lexical analysers
2. Formal token specification
3. Token recognition
4. Finite (State) Automata
5. Regular Expressions to Automata.
6. Lexical analyzer in HACS
First compilation phase

source program

<table>
<thead>
<tr>
<th>Characters</th>
<th>Tokens</th>
<th>Syntax Analysis</th>
<th>Symbol Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>Tree</td>
<td>Semantic Analysis</td>
<td></td>
</tr>
<tr>
<td>IR</td>
<td>IR</td>
<td>Intermediate Representation Generator</td>
<td></td>
</tr>
<tr>
<td>IR</td>
<td></td>
<td>Optimizer</td>
<td></td>
</tr>
<tr>
<td>IR</td>
<td></td>
<td>Code Generator</td>
<td></td>
</tr>
<tr>
<td>Machine code</td>
<td></td>
<td>Machine-Dependent Code Optimizer</td>
<td></td>
</tr>
</tbody>
</table>

target machine code
Purpose

Primary roles of the lexer/scanner:

- read input characters and expand macros,
- identify lexeme patterns for tokens,
- identify token attributes,
- construct the symbol table,
- error handling and reporting,
- strip whitespace and comments.
Software modules involved

Source Program

Lexical Analysis

Syntax Analysis

Semantic Analysis

Character Stream

Lexer/scanner

Parser

Syntax Tree

getNextChar

getNextToken

Char

Token
Question

Why separating lexical and syntactic analysis?

- Simplifies overall compiler design.
- Improves compiler efficiency (allows specialized techniques for lexer issues vs parser issues).
- Better portability (input-device specific peculiarities restricted to lexer).
Example

The stream of characters:

\[ \text{position} = \text{initial} + \text{rate} \times 60 \]

Scanned into list of tokens, one for each lexeme:

\[
\langle \text{id}, 1 \rangle \langle = \rangle \langle \text{id}, 2 \rangle \langle + \rangle \langle \text{id}, 3 \rangle \langle \ast \rangle \langle \text{num}, 60 \rangle
\]

\[
\begin{array}{|c|c|}
\hline
1 & \text{position} \\
2 & \text{initial} \\
3 & \text{rate} \\
\hline
\end{array}
\]
### Example

<table>
<thead>
<tr>
<th>Lexeme</th>
<th>Token</th>
<th>Lexeme pattern (informal)</th>
<th>Attribute value</th>
</tr>
</thead>
<tbody>
<tr>
<td>position</td>
<td>⟨id, 1⟩</td>
<td>identifier string</td>
<td>1</td>
</tr>
<tr>
<td>=</td>
<td>⟨=⟩</td>
<td>equality symbol</td>
<td></td>
</tr>
<tr>
<td>initial</td>
<td>⟨id, 2⟩</td>
<td>identifier string</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>⟨+⟩</td>
<td>addition symbol</td>
<td></td>
</tr>
<tr>
<td>rate</td>
<td>⟨id, 3⟩</td>
<td>identifier string</td>
<td>3</td>
</tr>
<tr>
<td>*</td>
<td>⟨*⟩</td>
<td>multiplication symbol</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>⟨num, 60⟩</td>
<td>numeric constant</td>
<td>60</td>
</tr>
</tbody>
</table>
Tokens

Typical classes of tokens in a programming language:

- keywords,
- operators,
- one token for identifiers,
- tokens for constants (numbers, literal strings,..),
- tokens for punctuation symbols (comma, semicolon,..).
Token attributes

Typical attribute values for the identifier token (id):

- the precise lexeme,
- its type/storage requirements,
- location where first found (error reporting),
Example

```c
float limitedSquare(float x) {
    /* returns x, squared, but never more than 100 */
    return (x<=-10.0||x>=10.0) ? 100 : x*x;
}
```

What tokens will likely be generated from the above C program?
Dealing with errors

Lexical analyser unable to proceed when no pattern matches

- Panic mode recovery: delete characters from input until a matching pattern is found.
- Insert missing character.
- Replace a character with another.
- Transpose two adjacent characters.
Input buffering

Buffering issues

- Lexer may need to look at least a character ahead to make a token decision.
- Example: Assume two tokens are defined as ‘*’ and ‘***’. Now consider the input character sequence ‘**a’. We need to scan and buffer all three characters to decide the tokens are two ‘*’. 
1. Lexical analysers
2. Formal token specification
3. Token recognition
4. Finite (State) Automata
5. Regular Expressions to Automata.
6. Lexical analyzer in HACS
Specifying tokens: lexeme patterns

Some lexeme patterns: +, 6-0, =, ==, p-o-s-i-t-i-o-n, ...

“The world” of lexeme patterns include finite, infinite, cyclic, and balanced patterns ...

partition into

- Regular expressions (RE) (formal token descriptions),
- other patterns.
Lexical analysers Formal token specification Token recognition Finite (State) Automata Regular Expressions to Automata. Lexical analyzer in HACS

Lexer Generator

- Lex, Flex
- HACS has its own lexical-analyser generator
Token specification

Identifiers as RE:

\[
\text{letter}(\text{letter} | \text{digit})^* 
\]

where

- \textit{letter} – any letter in \textit{the alphabet},
- \textit{digit} – any single digit from the 0 to 9 range.
Token specification

Some lexeme patterns/strings as RE:

- operator: +
- equality: (=|==)
- number: \{ d^+ | d \in Digit \}
- identifiers: \{ l(d|l)^* | d \in Digit, l \in Letter \}
- empty string: \( \epsilon \) (epsilon)

Digit and Letter stands for alphabets.
Definition

Regular Expression concepts:

**Alphabet**  any finite set of symbols.

**String**  finite sequence of symbols drawn from an alphabet.

**Language**  countable set of strings over some fixed alphabet.
## Definition

Basic RE taxonomy:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Syntax</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Character</td>
<td>e</td>
<td>b</td>
<td>b</td>
</tr>
<tr>
<td>Concatenation</td>
<td>$e_1 e_2$</td>
<td>$b c$</td>
<td>$b c$</td>
</tr>
<tr>
<td>Choice</td>
<td>$e_1</td>
<td>e_2$</td>
<td>$a</td>
</tr>
<tr>
<td>Repetition($\geq 0$)</td>
<td>$e^*$</td>
<td>$a^*$</td>
<td>$\epsilon, a, aa, \ldots$</td>
</tr>
<tr>
<td>Grouping</td>
<td>$(e)$</td>
<td>$(a</td>
<td>b)c$</td>
</tr>
</tbody>
</table>

where * denotes the *Kleene Closure* of the generated language.
Examples

Describe the languages (patterns/strings) generated by REs:

- $(\epsilon | a)b^*$
- $a(a|b)^*a$
- $a^*ba^*ba^*ba^*$
## Definition

Extended RE taxonomy:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Syntax</th>
<th>Pattern</th>
<th>Matches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optional</td>
<td>$e?$</td>
<td>$(a \mid b)?$</td>
<td>$\epsilon, a, b$</td>
</tr>
<tr>
<td>Repetition ($\geq 1$)</td>
<td>$e^+$</td>
<td>$a^+$</td>
<td>$a, aa, \ldots$</td>
</tr>
<tr>
<td>Character class</td>
<td>[...]</td>
<td>$[0-9 _ a-z]$</td>
<td>$7, _, c$</td>
</tr>
</tbody>
</table>

where $+$ denotes the *Positive Closure* of the generated language.
Examples

State REs for the following languages:

- all non-empty strings over the alphabet of ‘0’s and ‘1’s,
- all strings beginning with a digit, followed by either a or b, and ending with an uppercase letter.
Answers

- all non-empty strings over the alphabet of ‘0’s and ‘1’s,
- \((0|1)^+\)
- all strings beginning with a digit, followed by either a or b, and ending with an uppercase letter.
- \([0-9](a|b)[A-Z]\)
Simulation

Extended RE features as basic RE:

<table>
<thead>
<tr>
<th>Extended syntax</th>
<th>Basic syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>$e^?$</td>
<td>$e</td>
</tr>
<tr>
<td>$e^+$</td>
<td>$ee^*$</td>
</tr>
<tr>
<td>$[e_1 - e_2 \ldots e_i - e_{i+1}]$</td>
<td>$e^1</td>
</tr>
</tbody>
</table>

where $e^1 \in \{e_1, \ldots, e_2\}, \ldots, e^i \in \{e_i, \ldots, e_{i+1}\}, i \geq 1$
Token specification in HACS as REs:

```
space  [ \t\n] ;

token Int    | ⟨Digit⟩+ ;
token Float  | ⟨Int⟩ "." ⟨Int⟩ ;
token Id     | ⟨Lower⟩+ (\'_'? ⟨Int⟩)? ;

token fragment Digit  | [0–9] ;
token fragment Lower   | [a–z] ;
```
Token specification

Some common computer alphabets:

- Ascii code (e.g., \177)
- Unicode (e.g., \u12f)

Example REs from the real world (in HACS):

- [\000-\037] ascii control codes,
- [\u0-9a-fA-F]+ unicode escape characters.
Additional string vocabulary

Prefix  a string obtained by removing zero or more symbols from the end.

Suffix  a string obtained by removing zero or more symbols from the beginning.

Substring  a string obtained by removing any prefix and suffix.

Subsequence  a string formed by removing zero or more arbitrary positions.

Consider: ‘banana’
1. Lexical analysers
2. Formal token specification
3. Token recognition
4. Finite (State) Automata
5. Regular Expressions to Automata.
6. Lexical analyzer in HACS
### Formal Languages

**Sets of strings**

Approaches for defining formal languages (strings):

- basic RE taxonomy
- extended RE taxonomy
- context-free grammar (CFG)
- nondeterministic finite automaton
- deterministic finite automaton
Formal grammars

Grammar for branching statements

\[ stmt \rightarrow if \ expr \ then \ stmt \]
\[ \ | \ if \ expr \ then \ stmt \ else \ stmt \]
\[ \ | \ \epsilon \]
\[ expr \rightarrow term \ relop \ term \]
\[ \ | \ term \]
\[ term \rightarrow id \]
\[ \ | \ number \]
Formal grammars

Patterns for tokens

\[
\begin{align*}
  id & \rightarrow \text{letter(letter | digits)*} \\
  letter & \rightarrow [A - Z a - z] \\
  number & \rightarrow \text{digits(.digits)?([+-]?digits)?} \\
  digits & \rightarrow \text{digit}^* \\
  digit & \rightarrow [0 - 9] \\
  if & \rightarrow \text{if} \\
  then & \rightarrow \text{then} \\
  else & \rightarrow \text{else} \\
  relop & \rightarrow < | > | <= | >= | = | <> 
\end{align*}
\]
Diagram recognizes the `relop` token.

```
start \(\rightarrow 0 \rightarrow 1 \rightarrow \cdots \rightarrow 8 \rightarrow \text{return(relop, GT)}\)
```

- \(0 \rightarrow < \rightarrow 1 \rightarrow \text{return(relop, LE)}\)
- \(1 \rightarrow > \rightarrow 2 \rightarrow \text{return(relop, NE)}\)
- \(1 \rightarrow = \rightarrow 3 \rightarrow \text{return(relop, EQ)}\)
- \(1 \rightarrow > \rightarrow 4 \rightarrow \text{return(relop, LT)}\)
- \(5 \rightarrow = \rightarrow \text{return(relop, EQ)}\)
- \(6 \rightarrow = \rightarrow 7 \rightarrow \text{return(relop, GE)}\)
- \(5 \rightarrow > \rightarrow 7 \rightarrow \text{return(relop, GE)}\)
- \(8 \rightarrow * \rightarrow \text{return(relop, GT)}\)
Expressive Language Power

“Language power” given by the possibly generated strings

Comparisons:

- extended and basic RE has same power
- CFG more powerful than RE ...
- finite automata and RE has same power
Comparing CFG and RE

CFG at least as expressive as RE:
Can emulate RE with CFG:

<table>
<thead>
<tr>
<th>RE:</th>
<th>CFG:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a</td>
<td>b)* abb</td>
</tr>
<tr>
<td></td>
<td>Y → YZ</td>
</tr>
<tr>
<td></td>
<td>Z → a</td>
</tr>
</tbody>
</table>
Comparing CFG and RE

CFG more expressive than RE:

<table>
<thead>
<tr>
<th>CFG:</th>
<th>RE:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X \rightarrow [X]</td>
<td>a$</td>
</tr>
</tbody>
</table>

Captures a, [a], [[a]], ..., [$a]^n$. 
Lexical analysers

Formal token specification

Token recognition

Finite (State) Automata

Regular Expressions to Automata.

Lexical analyzer in HACS
Finite state automata are string recognizers: answering “yes” or “no” to each input string.

Nondeterministic finite automata (NFA) transition diagrams with no restrictions on the labels of their edges.

Deterministic automata transition diagrams where for each state and each symbol of the input alphabet there is exactly one edge leaving that state.
NFA: Nondeterministic Finite (State) Automata

NFA accepting $a(a|b)^*b$:

![Nondeterministic Finite Automata Diagram]
NFA accepting $a(a|b)^*b$

- States = \{1, 2, 3, 4\}
- $\Sigma = \{a, b\}$

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
<th>$\epsilon$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>{2}</td>
<td>{}</td>
<td>{}</td>
</tr>
<tr>
<td>2</td>
<td>{3}</td>
<td>{3, 4}</td>
<td>{}</td>
</tr>
<tr>
<td>3</td>
<td>{}</td>
<td>{4}</td>
<td>{2}</td>
</tr>
<tr>
<td>4</td>
<td>{}</td>
<td>{}</td>
<td>{}</td>
</tr>
</tbody>
</table>

- Start = 1
- Finals = \{4\}
NFA core definition

- finite set of states \( S \) (circles),
- set of input symbols \( \Sigma \) (labels),
- next-state transition function \( \tau : S \times \Sigma \to \Sigma \) (edges, table),
- dedicated start state \( s_0 \in S \) (start arrow),
- set of accepting states \( \subseteq S \) (double circles).
DFA: Deterministic Finite (State) Automata

**DFA accepting** $a(a|b)^*b$:

![DFA Diagram]
DFA accepting $a(a|b)^*b$

- **States** = \{A, B, C, D, E\}
- **Σ** = \{a, b\}

<table>
<thead>
<tr>
<th>State</th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>

- **Start** = A
- **Finals** = \{D\}
DFA core definition

Special case of NFA:

- no moves on $\epsilon$,
- for each $(s, a) \in S \times \Sigma : \exists \tau(s, a)$ unique!
- Transition graph: there is exactly one edge out of each state $s$ and each label $a$.
- Transition table: each entry is a single state.
## Some definitions

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\epsilon$-closure$(s)$</td>
<td>Set of NFA states reachable from set $s$ on $\epsilon$-transitions alone.</td>
</tr>
<tr>
<td>$\epsilon$-closure$(T)$</td>
<td>Set of NFA states reachable from some set $s$ in $T$ on $\epsilon$-transitions alone.</td>
</tr>
<tr>
<td>$move(T, a)$</td>
<td>Set of NFA states to which there is a transition on input symbol $a$ from some state $s$ in $T$</td>
</tr>
</tbody>
</table>
Subset construction

NFA to DFA (Algorithm 3.20.):

```plaintext
// Initially, $\epsilon$-closure($s_0$) is the only state in $Dstates$, and it is unmarked;
while (there is an unmarked state $T$ in $Dstates$) {
    mark $T$;
    for (each input symbol $a$) {
        $\bigcup$ = $\epsilon$-closure(move($T$, $a$));
        if ($\bigcup$ is not in $Dstates$)
            add $\bigcup$ as an unmarked state to $Dstates$
        $Dtran[T,a] = \bigcup$
    }
}
```

*Dstates* states of the DFA we are constructing.
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

$$\epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\} = A$$
NFA → DFA Translation for \((a|b)^*abb\)

\[ \epsilon\text{-closure}(0) = \{0, 1, 2, 4, 7\} = A \]
Lexical analysers  Formal token specification  Token recognition  Finite (State) Automata  Regular Expressions to Automata.  Lexical analyzer in HACS

NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

$\epsilon$-closure$(0) = \{0, 1, 2, 4, 7\} = A$

Eva Rose, Kristoffer Rose  Compiler Construction (CSCI-GA.2130-001) 2. Lexical Analysis  September 11, 2014  47 / 54
**NFA → DFA Translation for \((a|b)^*abb\)**

\[
DTran[A, a] = \epsilon\text{-closure}(move(A, a))
\]
\[
= \epsilon\text{-closure}(\{3, 8\})
\]
\[
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
\begin{align*}
DTran[A, a] &= \epsilon\text{-closure}(\text{move}(A, a)) \\
&= \epsilon\text{-closure}(\{3, 8\}) \\
&= \{1, 2, 3, 4, 6, 7, 8\} = B
\end{align*}
\]
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

\[ DTran[A, a] = \epsilon\text{-closure}(move(A, a)) \]
\[ = \epsilon\text{-closure}\{3, 8\} \]
\[ = \{1, 2, 3, 4, 6, 7, 8\} = B \]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[A, b] = \epsilon\text{-closure}(move(A, b))
= \epsilon\text{-closure}(\{5\})
= \{1, 2, 4, 5, 6, 7\} = C
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[A, b] = \epsilon\text{-closure}(\text{move}(A, b))
\]
\[
= \epsilon\text{-closure}(\{5\})
\]
\[
= \{1, 2, 4, 5, 6, 7\} = C
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[A, b] = \epsilon\text{-closure}(move(A, b)) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C
\]
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

$$DTran[B, a] = \epsilon\text{-closure}(\text{move}(B, a)) = \epsilon\text{-closure}([3, 8]) = [1, 2, 3, 4, 6, 7, 8] = B$$
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[B, a] = \epsilon\text{-closure}(\text{move}(B, a))
= \epsilon\text{-closure}\{3, 8\}
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
NFA \rightarrow DFA Translation for \((a|b)^*abb\)

\[
\begin{align*}
DTran[B, a] &= \epsilon\text{-closure}(move(B, a)) \\
&= \epsilon\text{-closure}\{3, 8\} \\
&= \{1, 2, 3, 4, 6, 7, 8\} = B
\end{align*}
\]
NFA → DFA Translation for $(a|b)^*abb$

\[ DTran[B, b] = \epsilon\text{-closure}(move(B, b)) \]
\[ = \epsilon\text{-closure}(\{5, 9\}) \]
\[ = \{1, 2, 4, 5, 6, 7, 9\} = D \]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[B, b] = \epsilon\text{-}closure(move(B, b))
\]
\[
= \epsilon\text{-}closure(\{5, 9\})
\]
\[
= \{1, 2, 4, 5, 6, 7, 9\} = D
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[B, b] = \epsilon\text{-closure}(move(B, b))
\]

\[
= \epsilon\text{-closure}\{5, 9\}
\]

\[
= \{1, 2, 4, 5, 6, 7, 9\} = D
\]
**NFA → DFA Translation for \((a|b)^*abb\)**

\[
\text{DTran}[C, a] = \epsilon\text{-closure}(\text{move}(C, a)) \\
= \epsilon\text{-closure}\{3, 8\} \\
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
D\text{Tran}[C, a] = \epsilon\text{-closure}(move(C, a))
\]
\[
= \epsilon\text{-closure}\{3, 8\}
\]
\[
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

$$DTran[C, a] = \epsilon\text{-closure}\left(move(C, a)\right)$$
$$= \epsilon\text{-closure}\left\{3, 8\right\}$$
$$= \{1, 2, 3, 4, 6, 7, 8\} = B$$
NFA → DFA Translation for \((a|b)^*abb\)

\[
D\text{Tran}[C, b] = \epsilon\text{-closure}(move(C, b)) = \epsilon\text{-closure}\{5\} = \{1, 2, 4, 5, 6, 7\} = C
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[C, b] = \epsilon\text{-closure}(move(C, b))
\]

\[
= \epsilon\text{-closure}([5])
\]

\[
= \{1, 2, 4, 5, 6, 7\} = C
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[C, b] = \epsilon\text{-closure}(\text{move}(C, b))
\]
\[
= \epsilon\text{-closure}(\{5\})
\]
\[
= \{1, 2, 4, 5, 6, 7\} = C
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
\begin{align*}
DTrans[D, a] &= \epsilon\text{-closure}(\text{move}(D, a)) \\
&= \epsilon\text{-closure}([3, 8]) \\
&= \{1, 2, 3, 4, 6, 7, 8\} = B
\end{align*}
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[ DTrans[D, a] = \epsilon\text{-closure}(\text{move}(D, a)) = \epsilon\text{-closure}(\{3, 8\}) = \{1, 2, 3, 4, 6, 7, 8\} = B \]
NFA → DFA Translation for \((a|b)^*abb\)

\[ DTran[D, a] = \epsilon\text{-closure}(move(D, a)) \]
\[ = \epsilon\text{-closure}\{3, 8\} \]
\[ = \{1, 2, 3, 4, 6, 7, 8\} = B \]
**NFA \rightarrow DFA Translation for (a|b)^*abb**

\[
DTran[D, b] = \epsilon\text{-closure}(\text{move}(D, b)) = \epsilon\text{-closure}\left(\{5, 10\}\right) = \{1, 2, 4, 5, 6, 7, 10\} = E
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
\begin{align*}
DTran[D, b] &= \epsilon\text{-closure}(\text{move}(D, b)) \\
&= \epsilon\text{-closure}(\{5, 10\}) \\
&= \{1, 2, 4, 5, 6, 7, 10\} = E
\end{align*}
\]
NFA → DFA Translation for \((a|b)^*abb\)

\[DTran[D, b] = \epsilon\text{-closure}(move(D, b)) = \epsilon\text{-closure}(\{5, 10\}) = \{1, 2, 4, 5, 6, 7, 10\} = E\]
NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[E, a] = \epsilon\text{-closure}(\text{move}(E, a))
= \epsilon\text{-closure}\{3, 8\}
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
**NFA → DFA Translation for \((a|b)^*abb\)**

\[
DTran[E, a] = \epsilon\text{-closure}(move(E, a)) \\
= \epsilon\text{-closure}([3, 8]) \\
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

\[
DTran[E, a] = \epsilon\text{-closure}(move(E, a)) \\
= \epsilon\text{-closure}([3, 8]) \\
= \{1, 2, 3, 4, 6, 7, 8\} = B
\]
**NFA → DFA Translation for** (a|b)*abb

\[ DTran[E, b] = \epsilon\text{-closure}(move(E, b)) = \epsilon\text{-closure}\{5\} = \{1, 2, 4, 5, 6, 7\} = C \]
Lexical analysers
Formal token specification
Token recognition
Finite (State) Automata
Regular Expressions to Automata
Lexical analyzer in HACS

NFA → DFA Translation for \((a|b)^*abb\)

\[
DTran[E, b] = \epsilon\text{-closure}(move(E, b)) = \epsilon\text{-closure}(\{5\}) = \{1, 2, 4, 5, 6, 7\} = C
\]
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$

$DTran[E, b] = \epsilon$-closure($move(E, b)$)

$= \epsilon$-closure($\{5\}$)

$= \{1, 2, 4, 5, 6, 7\} = C$
NFA $\rightarrow$ DFA Translation for $(a|b)^*abb$
Exercise for NFA accepting $a(a|b)^*b$

Create DFA by subset construction:
Comparing DFA and NFA properties

<table>
<thead>
<tr>
<th></th>
<th>DFA</th>
<th>NFA</th>
</tr>
</thead>
<tbody>
<tr>
<td>labels from same state</td>
<td>unique</td>
<td>may be incomplete</td>
</tr>
<tr>
<td>$\epsilon$-labels</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>transition table</td>
<td>states</td>
<td>state sets</td>
</tr>
<tr>
<td>expressive power</td>
<td>same as RE</td>
<td>same as RE</td>
</tr>
<tr>
<td>trade-offs</td>
<td>easy to run</td>
<td>easy to design</td>
</tr>
</tbody>
</table>
1. Lexical analysers
2. Formal token specification
3. Token recognition
4. Finite (State) Automata
5. Regular Expressions to Automata.
6. Lexical analyzer in HACS
Definition: RE $\rightarrow$ NFA translation

Constructing NFA from RE (Algorithm 3.23): Regular expression $r$ to NFA with initial/final states $i, f$:

$$[r]_{i.f} = \begin{array}{c}
\text{i} \\
\text{r} \\
\text{f}
\end{array}$$
Definition: RE → NFA translation

\[
[a]_{if} \equiv \begin{array}{c}
\text{\(i\)} \\
\text{\(\rightarrow\)}
\end{array}
\begin{array}{c}
\text{\(a\)} \\
\end{array}
\begin{array}{c}
\text{\(f\)} \\
\end{array}
\]

\[
[st]_{if} \equiv \begin{array}{c}
\text{\(i\)} \\
\text{\(\rightarrow\)}
\end{array}
\begin{array}{c}
\text{\(s\)} \\
\text{\(f_s\)}
\end{array}
\begin{array}{c}
\epsilon \\
\end{array}
\begin{array}{c}
\text{\(i_t\)} \\
\text{\(t\)} \\
\text{\(f\)}
\end{array}
\]

\[
[s|t]_{if} \equiv \begin{array}{c}
\text{\(i\)} \\
\text{\(\rightarrow\)}
\end{array}
\begin{array}{c}
\text{\(s\)} \\
\text{\(f_s\)}
\end{array}
\begin{array}{c}
\epsilon \\
\end{array}
\begin{array}{c}
\text{\(i_t\)} \\
\text{\(t\)} \\
\text{\(f\)}
\end{array}
\]

\[
[s^*]_{if} \equiv \begin{array}{c}
\text{\(i\)} \\
\text{\(\rightarrow\)}
\end{array}
\begin{array}{c}
\text{\(s\)} \\
\text{\(f_s\)}
\end{array}
\begin{array}{c}
\epsilon \\
\end{array}
\begin{array}{c}
\text{\(f\)}
\end{array}
\]

\[
[\epsilon]_{if} \equiv \begin{array}{c}
\text{\(i\)} \\
\text{\(\rightarrow\)}
\end{array}
\begin{array}{c}
\epsilon \\
\end{array}
\begin{array}{c}
\text{\(f\)}
\end{array}
\]
Exercise: RE $\rightarrow$ NFA translation

Construct an NFA that accepts the same language as $(a|b)^*abb$ using Algorithm 3.23.
Lexical analysers

Formal token specification

Token recognition

Finite (State) Automata

Regular Expressions to Automata.

Lexical analyzer in HACS