Compiler Design

Lecture 4: Automatic Lexer Generation
(EaC§2.4)

Christophe Dubach
15 January 2021
Table of contents

Finite State Automata for Regular Expression
  Finite State Automata
  Non-determinism

From Regular Expression to Generated Lexer
  Regular Expression to NFA
  From NFA to DFA

Final Remarks
Starting from a collection of regular expressions (RE) we automatically generate a Lexer.

- We use *finite state automata* (FSA) for the construction
Finite State Automata for Regular Expression
Finite State Automata for Regular Expression

Finite State Automata
Definition: finite state automata

A finite state automata is defined by:

- $S$, a finite set of states
- $\Sigma$, an alphabet, or character set used by the recogniser
- $\delta(s, c)$, a transition function
  (takes a state and a character as input, and returns new state)
- $s_0$, the initial or start state
- $S_F$, a set of final states (a stream of characters is accepted if the automata ends up in a final state)
Example: register names

\[ \text{register} ::= 'r' ( '0' | '1' | \ldots | '9' ) ( '0' | '1' | \ldots | '9' )^* \]

The RE (Regular Expression) corresponds to a recogniser (or finite state automata):

![Finite State Automata for Regular Expression](image)
Finite State Automata (FSA) operation:

- Start in state $s_0$ and take transitions on each input character
- The FSA accepts a word $x$ iff $x$ leaves it in a final state ($s_2$)

Examples:

- $r17$ takes it through $s_0, s_1, s_2$ and accepts
- $r$ takes it through $s_0, s_1$ and fails
- $a$ starts in $s_0$ and leads straight to failure
Table encoding and skeleton code

To be useful a recogniser must be turned into code

\[ \begin{array}{c|c|c|c} \delta & \text{'}r\text{'} & \text{'}0\text{'|'}1\text{'|}...\text{'9'} & \text{others} \\ \hline s_0 & s_1 & \text{error} & \text{error} \\ s_1 & \text{error} & s_2 & \text{error} \\ s_2 & \text{error} & s_2 & \text{error} \end{array} \]

**Skeleton recogniser**

\[
\begin{align*}
c & = \text{next character} \\
\text{state} & = s_0 \\
\text{while} (c \neq \text{EOF}) \\
& \quad \text{state} = \delta(\text{state}, c) \\
& \quad c = \text{next character} \\
\text{if} (\text{state final}) \\
& \quad \text{return success} \\
\text{else} \\
& \quad \text{return error}
\end{align*}
\]
Finite State Automata for Regular Expression

Non-determinism
Deterministic Finite Automaton
Each RE corresponds to a Deterministic Finite Automaton (DFA). However, it might be hard to construct directly.

What about an RE such as \((a|b)^*abb\) ?

This is a little different:

- \(s_0\) has a transition on \(\epsilon\), which can be followed without consuming an input character
- \(s_1\) has two transitions on \(a\)
- This is a Non-deterministic Finite Automaton (NFA)
Non-deterministic vs deterministic finite automata

**Deterministic finite state automata (DFA):**
- All edges leaving the same node have distinct labels
- There is no $\epsilon$ transition

**Non-deterministic finite state automata (NFA):**
- Can have multiple edges with same label leaving from the same node
- Can have $\epsilon$ transition
- This means we might have to backtrack

Backtracking example for a NFA: input = aabb

![Diagram of NFA with states and transitions](image-url)
From Regular Expression to Generated Lexer
Automatic Lexer Generation

It is possible to systematically generate a lexer for any regular expression. This can be done in three steps:

1. regular expression (RE) $\rightarrow$ non-deterministic finite automata (NFA)
2. NFA $\rightarrow$ deterministic finite automata (DFA)
3. DFA $\rightarrow$ generated lexer
From Regular Expression to Generated Lexer

Regular Expression to NFA
1st step: RE → NFA (Ken Thompson, CACM, 1968)

“x”

\[ \begin{align*}
s_0 & \xrightarrow{x} s_1 \\
M & \end{align*} \]

\[ \begin{align*}
M & \\
\end{align*} \]

\[ \begin{align*}
M | N & \\
\end{align*} \]

\[ \begin{align*}
M^+ & \\
\end{align*} \]
Example: $a(b|c)^*$
Example: $a(b|c)^*$

A human would do: $s_0 \xrightarrow{a} s_1$
From Regular Expression to Generated Lexer

From NFA to DFA
Executing a non-deterministic finite automata requires backtracking, which is inefficient. To overcome this, we need to construct a DFA from the NFA.

The main idea:

- We build a DFA which has one state for each set of states the NFA could end up in.
- A set of state is final in the DFA if it contains the final state from the NFA.
- Since the number of states in the NFA is finite \( (n) \), the number of possible sets of states is also finite (maximum \( 2^n \), hint: state encoded as binary vectors).
Assuming the state of the NFA are labelled $s_i$ and the states of the DFA we are building are labelled $q_i$.

We have two key functions:

- $\text{reachable}(s_i, \alpha)$ returns the set of states reachable from $s_i$ by consuming character $\alpha$
- $\epsilon$-closure($s_i$) returns the set of states reachable from $s_i$ by $\epsilon$ (e.g. without consuming a character)
### The Subset Construction algorithm (Fixed point iteration)

\[ q_0 = \epsilon\text{-closure}(s_0); \ Q = \{q_0\}; \ \text{add} \ q_0 \ \text{to WorkList} \]

**while** (WorkList not empty)

- remove \( q \) from WorkList
- for each \( \alpha \in \Sigma \)
  - \( \text{subset} = \epsilon\text{-closure(} \text{reachable}(q, \alpha) \text{)} \)
  - \( \delta(q, \alpha) = \text{subset} \)
  - if (subset \( \notin Q \) then
    - add subset to \( Q \) and to WorkList

### The algorithm (in English)

- Start from start state \( s_0 \) of the NFA, compute its \( \epsilon \)-closure
- Build subset from all states reachable from \( q_0 \) for character \( \alpha \)
- Add this subset to the transition table/function \( \delta \)
- If the subset has not been seen before, add it to the worklist
- Iterate until no new subset are created
Informal proof of termination

- Q contains no duplicates (test before adding)
- similarly we will never add twice the same subset to the worklist
- bounded number of states; maximum $2^n$ subsets, where $n$ is number of state in NFA

$\Rightarrow$ the loop halts
Informal proof of termination

- $Q$ contains no duplicates (test before adding)
- similarly we will never add twice the same subset to the worklist
- bounded number of states; maximum $2^n$ subsets, where $n$ is number of state in NFA

$\Rightarrow$ the loop halts

End result

- $S$ contains all the reachable NFA states
- It tries each symbol in each $s_i$
- It builds every possible NFA configuration

$\Rightarrow$ $Q$ and $\delta$ form the DFA
NFA → DFA

\[ a(b|c)^* \]

\[
\begin{array}{cccc}
\text{NFA states} & a & b & c \\
q_0 & s_0 & & \\
\end{array}
\]

**ε-closure(reachable(q, α))**
\( a(b|c)^* \)
$a(b|c)^*$

$$
\begin{array}{c|ccc}
\text{NFA states} & a & b & c \\
\hline
q_0 & s_0 & q_1 \\
q_1 & s_1, s_2, s_3, s_4, s_6, s_9 \\
\end{array}
$$

\text{$\epsilon$-closure(reachable($q, \alpha$))}
NFA → DFA

\[ a(b|c)^* \]

\[ \epsilon\text{-closure}(reachable(q, \alpha)) \]

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( s_0 )</td>
<td>( q_1 )</td>
<td>none</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( s_1, s_2, s_3, )</td>
<td>( s_4, s_6, s_9 )</td>
<td></td>
</tr>
</tbody>
</table>
$a(b|c)^*$
$a(b|c)^*$

### $\varepsilon$-closure(reachable($q, \alpha$))

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$s_0$</td>
<td>$q_1$</td>
<td>none</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s_1, s_2, s_3, s_4, s_6, s_9$</td>
<td>none</td>
<td></td>
</tr>
</tbody>
</table>

NFA $\rightarrow$ DFA
\(a(b|c)^*\)

### NFA states and \(\epsilon\)-closure

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0) (s_0)</td>
<td>(q_1)</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>(q_1) (s_1, s_2, s_3, s_4, s_6, s_9)</td>
<td>none</td>
<td>(q_2)</td>
<td></td>
</tr>
<tr>
<td>(q_2)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
$a(b|c)^*$

### $\epsilon$-closure($\text{reachable}(q, \alpha)$)

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$s_0$</td>
<td>$q_1$</td>
<td>none</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s_1, s_2, s_3,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$s_4, s_6, s_9$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s_5, s_8, s_9,$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\( a(b|c)^* \)

### \( \varepsilon \)-closure(\( \text{reachable}(q, \alpha) \))

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( s_0 )</td>
<td>( q_1 )</td>
<td>none</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( s_1, s_2, s_3, s_4, s_6, s_9 )</td>
<td>none</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( s_5, s_8, s_9, s_3, s_4, s_6 )</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>
$a(b|c)^*$

### NFA states

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$s_0$</td>
<td>$q_1$</td>
<td>none</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s_1, s_2, s_3,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$s_4, s_6, s_9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s_5, s_8, s_9,$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$s_7, s_8, s_9,$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### ε-closure(reachable($q, \alpha$))
$a(b|c)^*$

$\epsilon\text{-closure}(\text{reachable}(q, \alpha))$

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$s_0$</td>
<td>$q_1$</td>
<td>none</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s_1$, $s_2$, $s_3$, $s_4$, $s_6$, $s_9$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s_5$, $s_8$, $s_9$, $s_3$, $s_4$, $s_6$</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$s_7$, $s_8$, $s_9$, $s_3$, $s_4$, $s_6$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
\( a(b|c)^* \)

### \( \varepsilon\text{-closure}(\text{reachable}(q, \alpha)) \)

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>( q_0 )</td>
<td>( s_0 )</td>
<td>( q_1 )</td>
<td>none</td>
</tr>
<tr>
<td>( q_1 )</td>
<td>( s_1, s_2, s_3, s_4, s_6, s_9 )</td>
<td>none</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_2 )</td>
<td>( s_5, s_8, s_9, s_3, s_4, s_6 )</td>
<td>none</td>
<td>( q_2 )</td>
</tr>
<tr>
<td>( q_3 )</td>
<td>( s_7, s_8, s_9, s_3, s_4, s_6 )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
$a(b|c)^*$

\[ NFA \rightarrow DFA \]

\[
\begin{array}{cccc}
\text{NFA states} & a & b & c \\
q_0 & s_0 & q_1 & \text{none} & \text{none} \\
q_1 & s_1, s_2, s_3, s_4, s_6, s_9 & \text{none} & q_2 & q_3 \\
q_2 & s_5, s_8, s_9, s_3, s_4, s_6 & \text{none} & q_2 & q_3 \\
q_3 & s_7, s_8, s_9, s_3, s_4, s_6 & & & \\
\end{array}
\]

\[ \epsilon\text{-closure}(reachable(q, \alpha)) \]
$a(b|c)^*$

**NFA states**

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$s_0$</td>
<td>$q_1$</td>
<td>none</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s_1, s_2, s_3,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$s_4, s_6, s_9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s_5, s_8, s_9,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$s_7, s_8, s_9,$</td>
<td>none</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NFA → DFA

\(a(b|c)^*\)

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>(q_0)</td>
<td>(s_0)</td>
<td>(q_1)</td>
<td>none</td>
</tr>
<tr>
<td>(q_1)</td>
<td>(s_1, s_2, s_3,)</td>
<td>none</td>
<td>(q_2)</td>
</tr>
<tr>
<td></td>
<td>(s_4, s_6, s_9)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(q_2)</td>
<td>(s_5, s_8, s_9,)</td>
<td>none</td>
<td>(q_2)</td>
</tr>
<tr>
<td></td>
<td>(s_3, s_4, s_6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(q_3)</td>
<td>(s_7, s_8, s_9,)</td>
<td>none</td>
<td>(q_2)</td>
</tr>
<tr>
<td></td>
<td>(s_3, s_4, s_6)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\(\epsilon\)-closure(reachable(q, \(\alpha\)))
### NFA $\rightarrow$ DFA

#### $a(b|c)^*$

![Diagram of NFA and DFA](chart.png)

<table>
<thead>
<tr>
<th>NFA states</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$s_0$</td>
<td>$q_1$</td>
<td>none</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$s_1, s_2, s_3,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$s_4, s_6, s_9$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_2$</td>
<td>$s_5, s_8, s_9,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$q_3$</td>
<td>$s_7, s_8, s_9,$</td>
<td>none</td>
<td>$q_2$</td>
</tr>
<tr>
<td></td>
<td>$s_3, s_4, s_6$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Resulting DFA for $a(b|c)^*$

- Smaller than the NFA
- All transitions are deterministic (no need to backtrack!)
- Could be even smaller
  (see EaC§2.4.4 Hopcroft's Algorithm for minimal DFA)
- Can generate the lexer using skeleton recogniser seen earlier
Final Remarks
What can be so hard?

Poor language design can complicate lexing:

- **PL/I** does not have reserved words (keywords):
  
  ```plaintext
  if (cond) then then = else; else else = then
  ```

- In **Fortran & Algol68** blanks (whitespaces) are insignificant:
  
  ```plaintext
  do 10 i = 1.25 ≅ do 10 i = 1.25 (loop, 10 is statement label)
  do 10 i = 1.25 ≅ do10i = 1.25 (assignment)
  ```

- In **C, C++, Java** string constants can have special characters:
  newline, tab, quote, comment delimiters, . . .
Good language design makes lexing simpler:

- e.g. identifier cannot start with a digit in most modern languages
  ⇒ when we see a digit, it can only be the start of a number!

What does a C lexer sees?

```c
u24;  // identifier u24
24;   // signed number 24
24u;  // unsigned number 24
```
Building Lexer

The important point:

- All this technology lets us automate lexer construction
- Implementer writes down regular expressions
- Lexer generator builds NFA, DFA and then writes out code
- This reliable process produces fast and robust lexers

For most modern language features, this works:

- As a language designer you should think twice before introducing a feature that defeats a DFA-based lexer
- The ones we have seen (e.g. insignificant blanks, non-reserved keywords) have not proven particularly useful or long lasting
Parsing:

- Context-Free Grammars
- Dealing with ambiguity
- Recursive descent parser