Compiling Techniques
Lecture 4: Automatic Lexer Generation
Starting from a collection of regular expressions (RE) we automatically generate a Lexer.

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

Example: register names

register ::= ‘r’ (‘0’|’1’|...|’9’) (‘0’|’1’|...|’9’)*

The RE (Regular Expression) corresponds to a recognizer (or a finite state automata):
Table encoding and skeleton code

To be useful a recognizer must be turned into code

\[ r' \mid '0'| '1'|...|'9' \]

**Skeleton recogniser**

\[
c = \text{next_character()}
\]

\[
\text{state} = "s0"
\]

\[
\text{while } c := \text{EOF}:
\]

\[
\quad \text{state} = \delta(s, c)
\]

\[
\quad c = \text{next_character()}
\]

\[
\text{if (state final):}
\]

\[
\quad \text{return success}
\]

\[
\text{else:}
\]

\[
\quad \text{return error}
\]
Non-Determinism

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

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

- \(s0\) has a transition on \(\varepsilon\), which can be followed without consuming an input character.
- \(s1\) 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 transition

Non-deterministic finite state automata (NFA):

- Can have multiple edges with the same label leaving from the same node
- Can have $\epsilon$ transition

This means we might have to backtrack
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) → non-deterministic finite automata (NFA)
2. NFA → deterministic finite automata (DFA)
3. DFA → generated lexer
1st step: RE → NFA (Ken Thompson, CACM, 1968)

'x'

\[ M \mid N \]

\[ s0 \rightarrow s1 \]

\[ s0 \rightarrow s1 \]

\[ s1 \rightarrow s2 \]

\[ s2 \rightarrow s3 \]

\[ s3 \rightarrow s4 \]

\[ s4 \rightarrow s5 \]

\[ s5 \rightarrow s1 \]

\[ s1 \rightarrow s2 \]

\[ s2 \rightarrow s3 \]

\[ s3 \rightarrow s4 \]

\[ s4 \rightarrow s5 \]

\[ s5 \rightarrow s1 \]
1st step: RE → NFA (Ken Thompson, CACM, 1968)
Step 2: NFA → 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).
From NFA to DFA

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 $\alpha$
- $\text{closure}(s_i)$ returns the set of states reachable from $s_i$ by (e.g. without consuming a character)
Algorithm

The Subset Construction algorithm (Fixed point iteration)

\[ q_0 = \epsilon\text{-closure}(s_0); \quad Q = \{q_0\}; \quad \text{add } q_0 \text{ to WorkList} \]
\[ \text{while (WorkList not empty)} \]
\[ \quad \text{remove } q \text{ from WorkList} \]
\[ \quad \text{for each } \alpha \in \Sigma \]
\[ \quad \text{subset} = \epsilon\text{-closure}(\text{reachable}(q, \alpha)) \]
\[ \quad \delta(q, \alpha) = \text{subset} \]
\[ \quad \text{if } (\text{subset} \notin Q) \text{ then} \]
\[ \quad \text{add subset to } Q \text{ and to WorkList} \]

The algorithm (in English)

- Start from start state \( s_0 \) of the NFA, compute its \( \epsilon\text{-closure} \)
- 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
NFA for $a(b|c)^*$

**Table:**

<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>none</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>none</td>
<td>$q_2$</td>
</tr>
</tbody>
</table>
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
What can be so hard

Poor language design can complicate lexing

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

- In Fortran & Algol68 blanks (whitespaces) are insignificant:
  
  ```
  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, . . .
Building a 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
Next Lecture

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