Compiler Design

Lecture 4: Automatic Lexer Generation (EaC§2.4)

Christophe Dubach
Winter 2023
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 iif the automata ends up in a final state)
Example: register names

```
register ::= 'r' ( '0' | '1' | ... | '9' ) ( '0' | '1' | ... | '9' ) *
```

The RE (Regular Expression) corresponds to a recogniser (or finite state automata):
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
To be useful a recogniser must be turned into code

Table encoding RE

| $\delta$ | 'r' | '0'|'1'|...|'9' | others |
|----------|-----|-----|-----|-----|-------|
| $s_0$    | $s_1$ | error | error |
| $s_1$    | error | $s_2$ | error |
| $s_2$    | error | $s_2$ | error |

Skeleton recogniser

```plaintext
c = next character
state = s₀
while(c ≠ EOF)
    state = $\delta$(state, c)
    c = next character
if (state final)
    return success
else
    return error
```
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
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"

[M]

M|N

M N

M*

M+
Example: $a(b|c)^*$

A human would do: $s_0 \xrightarrow{a} s_1$

(automatic minimization possible)
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 want 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 (i.e. powerset) is also finite:
  - maximum $2^n$ (hint: set 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:

- \texttt{reachable}(s_i, \alpha) returns the set of states reachable from $s_i$ by consuming character $\alpha$
- \texttt{$\epsilon$-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)

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

while (WorkList not empty)
  remove $q$ from WorkList
  for each $\alpha \in \Sigma$
    $\text{subset} = \epsilon$-closure($\text{reachable}(q, \alpha)$)
    $\delta(q, \alpha) = \text{subset}$
    if ($\text{subset} \not\in Q$) then
      add $\text{subset}$ 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

⇒ 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

⇒ Q and $\delta$ form the 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>none</td>
<td>$q_2$</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>none</td>
<td>$q_2$</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>none</td>
<td>$q_2$</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?

Language design choice can complicate lexing:

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

  where are the variables?

- 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, ...
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?

```
u24;  // identifier u24
24;   // signed number 24
24u;  // unsigned number 24
```
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
Lexer generators input example


("["| ":") { count(); return ("["'); }

Wait a minute, what’s going on here??
Next lecture

Parsing:

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