Compiling Techniques
Lecture 4: Automatic Lexer Generation
(EaC§2.4)

Christophe Dubach

25 September 2018
Reminder

Action

Give us permission to access your gitlab repository.
Table of contents

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

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

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

We use finite state automata (FSA) for the construction
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 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)
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

| δ  | 'r' | '0'|'1'|...|'9' | others |
|----|-----|-----|-----|-----|--------|
| s₀ | s₁  | error | error |
| s₁ | error | s₂  | error |
| s₂ | error | s₂  | error |

Skeleton recogniser

c = next character
state = s₀
while (c ≠ EOF)
  state = δ(state, c)
c = next character
if (state final) return success
else return error
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 \((a|b)^*abb\) ?

![Diagram of a DFA]

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 the same label leaving from the same node
- Can have $\epsilon$ transition
- This means we might have to backtrack
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
1st step: RE → NFA (Ken Thompson, CACM, 1968)

```
\[
\begin{align*}
\text{"x"} & \\
\overset{\epsilon}{M} & \\
M | N & \\
M^+ & \\
\end{align*}
\]
```
Example: $a(b|c)^*$

A human would do:

```
\[
\begin{array}{c}
S_0 \\
\rightarrow \\
S_1
\end{array}
\Rightarrow
\begin{array}{c}
S_0 \\
\rightarrow \\
S_1
\end{array}
\]
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).
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 $\alpha$
- $\epsilon$-$\text{closure}(s_i)$ returns the set of states reachable from $s_i$ by $\epsilon$ (e.g. without consuming a character)
The Subset Construction algorithm (Fixed point iteration)

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

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

The algorithm (in English)

- Start from start state \( s_0 \) of the NFA, compute its \( \epsilon \)-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

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
a(b|c)*

NFA → DFA

\[
\begin{array}{c|cccc}
\text{NFA states} & \text{a} & \text{b} & \text{c} \\
\hline
q_0 & s_0 & & & \\
q_1 & s_1, s_2, s_3, s_4, s_6, s_9 & \text{none} & \text{q}_2 & \text{q}_3 \\
q_2 & s_5, s_8, s_9, s_3, s_4, s_6 & \text{none} & \text{q}_2 & \text{q}_3 \\
q_3 & s_7, s_8, s_9, s_3, s_4, s_6 & \text{none} & \text{q}_2 & \text{q}_3 \\
\end{array}
\]
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
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, . . .

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
Next lecture

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