Compiler Design

Lecture 4: Automatic Lexer Generation (EaC§2.4)

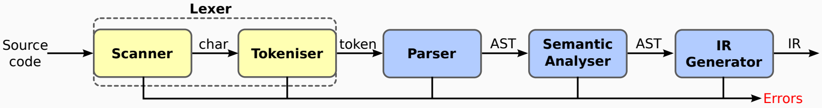
Christophe Dubach

Winter 2024

Timestamp: 2024/01/16 14:27:00

Table of contents

Automatic Lexer Generation



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

Idea: use a *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:

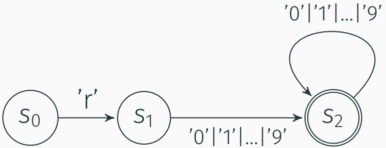
- S, a finite set of states
- Σ, an alphabet, or character set used by the recogniser
- $\delta(s, c)$, a transition function
  (takes a state and a character as input, and returns new state)
- s_0, the initial or start state
- S_F, a set of final states (a stream of characters is accepted iif the automata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

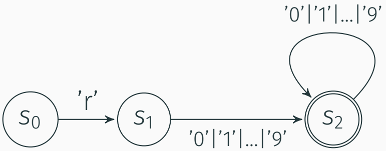
Finite State Automata for Regular Expression

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

Table encoding and skeleton code

To be useful a recogniser must be turned into code

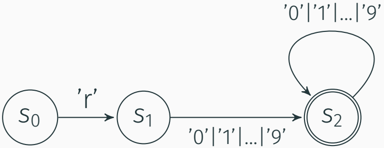


Table encoding RE

δ	'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

```
c = next character
state = s0
while(c ≠ EOF)
  state = δ(state, c)
  c = next character
if (state final)
  return success
else
  return error
```

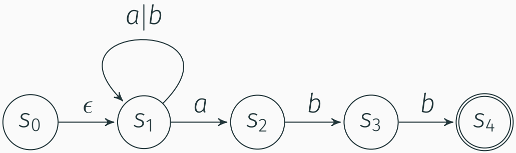
Finite State Automata for Regular Expression

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 ϵ, which can be followed without consuming an input character
- s_1 has two transitions on a
- This is a **Non-determinisitic Finite Automaton (NFA)**

Non-deterministic vs deterministic finite automata

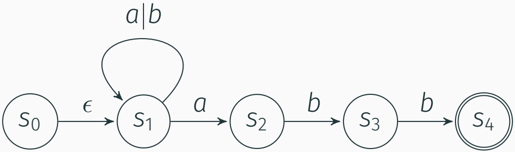
Deterministic finite state automata (DFA):

- All edges leaving the same node have distinct labels
- There is no ϵ transition

Non-deterministic finite state automata (NFA):

- Can have multiple edges with same label leaving the same node
- Can have ϵ 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.

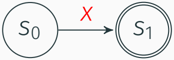
This can be done in three steps:

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

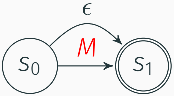
Regular Expression to NFA

"x"



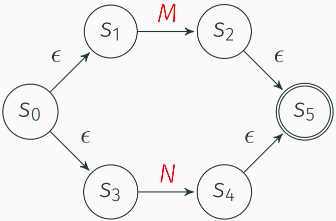
S_0
x
S_1

[M]



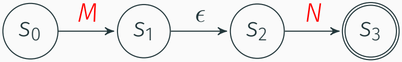
ϵ
M
S_0
S_1

M|N



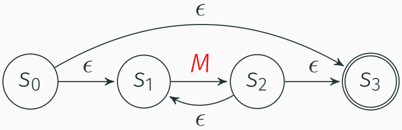
S_1
M
S_2
ϵ
ϵ
S_0
S_5
ϵ
ϵ
S_3
N
S_4

M N



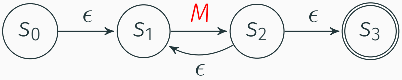
S_0
M
S_1
ϵ
S_2
N
S_3

*M**



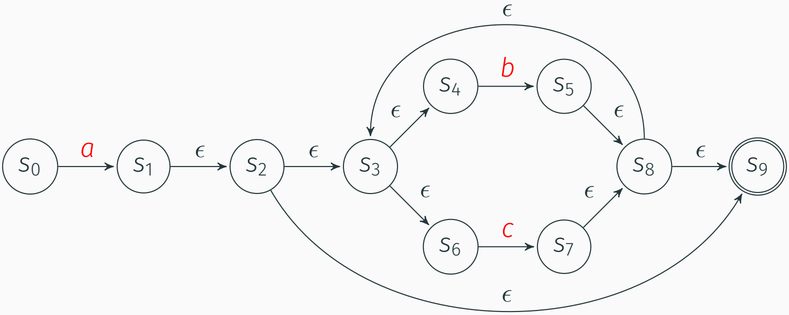
ϵ
S_0
ϵ
S_1
M
S_2
ϵ
S_3
ϵ

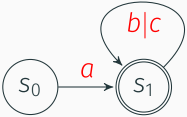
M+



S_0
ϵ
S_1
M
S_2
ϵ
S_3
ϵ

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



A human would do: 

(automatic minimization possible)

From Regular Expression to Generated Lexer

From NFA to DFA

Step 2: NFA $\rightarrow$ 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:

- reachable(s_i, α) returns the set of states reachable from s_i by consuming character α
- ϵ-closure(s_i) returns the set of states reachable from s_i by ϵ (*e.g.* without consuming a character)

The Subset Construction algorithm (Fixed point iteration)

```
q0 = ε-closure(s0); Q = {q0}; add q0 to WorkList
while (WorkList not empty)
  remove q from WorkList
  for each α ∈ Σ
    subset = ε-closure(reachable(q, α))
    δ(q, α) = subset
    if (subset ∉ Q) then
      add subset to Q and to WorkList
```

The algorithm (in English)

- Start from start state s_0 of the NFA, compute its ϵ-closure
- Build subset from all states reachable from q_0 for character α
- Add this subset to the transition table/function δ
- 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**

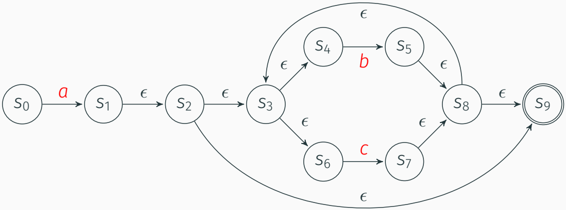
End result

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

$\Rightarrow$ **Q and δ form the DFA**

NFA → DFA

$a(b|c)^*$



		ϵ-*closure*(*reachable*(q, α))		
	NFA states	a	b	c
q_0	s_0	q_1	none	none
q_1	$s_1, s_2, s_3,$ s_4, s_6, s_9	none	q_2	q_3
q_2	$s_5, s_8, s_9,$ s_3, s_4, s_6	none	q_2	q_3
q_3	$s_7, s_8, s_9,$ s_3, s_4, s_6	none	q_2	q_3

Resulting DFA for $a(b|c)^*$

Graph

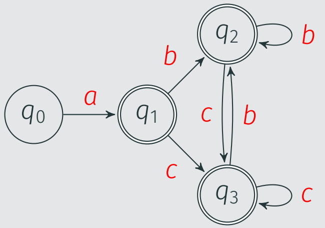


Table encoding

	a	b	c
q_0	q_1	error	error
q_1	error	q_2	q_3
q_2	error	q_2	q_3
q_3	error	q_2	q_3

- 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` $\cong$ `do 10 i = 1,25` (loop, 10 is statement label)

  `do 10 i = 1.25` $\cong$ `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
  $\Rightarrow$ 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
```

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

Lexer generators input

Example: ANSI-C grammar for tokens

https://www.cs.mcgill.ca/~cs520/2022/resources/ANSI-C-grammar-l.html

For instance:

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

Next lecture

Parsing:

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