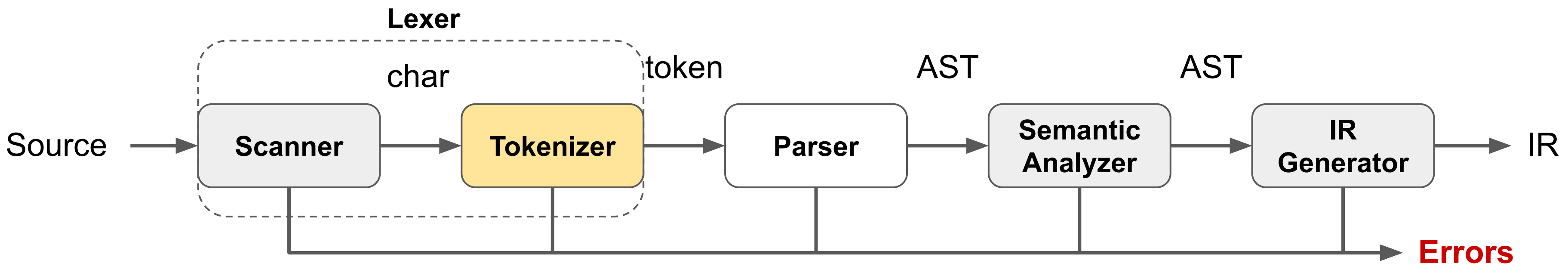
Compiling Techniques

Lecture 4: Automatic Lexer Generation

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
- **Σ**, an alphabet, or character set used by the recogniser
- **δ(s, c)**, a transition function (takes a state and a character and returns new state)
- **s0**, the initial or start state
- **SF**, 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 recognizer (or a finite state automata):

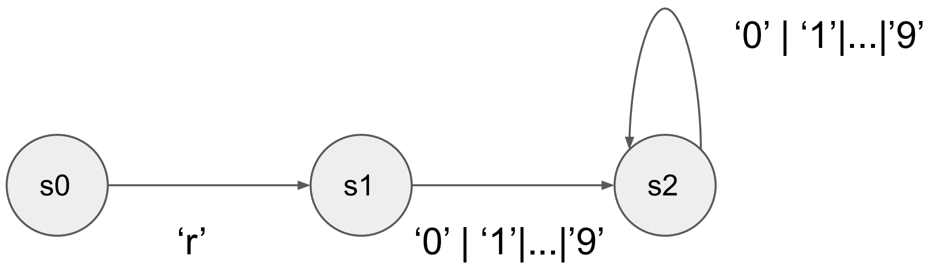
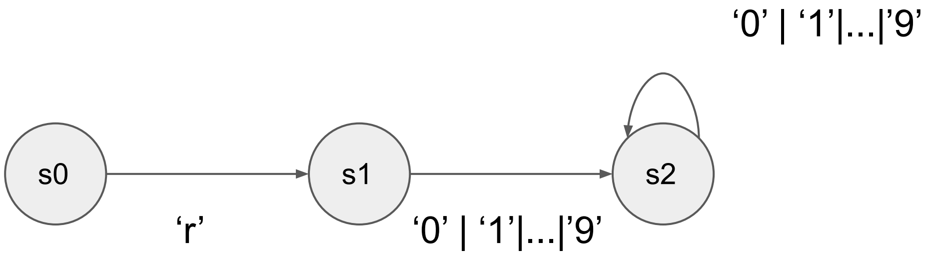


Table encoding and skeleton code

To be useful a recognizer must be turned into code



δ	‘r’	‘0’|’1’|...|’9’|	others
s0	s1	error	error
s1	error	s2	error
s2	error	s2	error

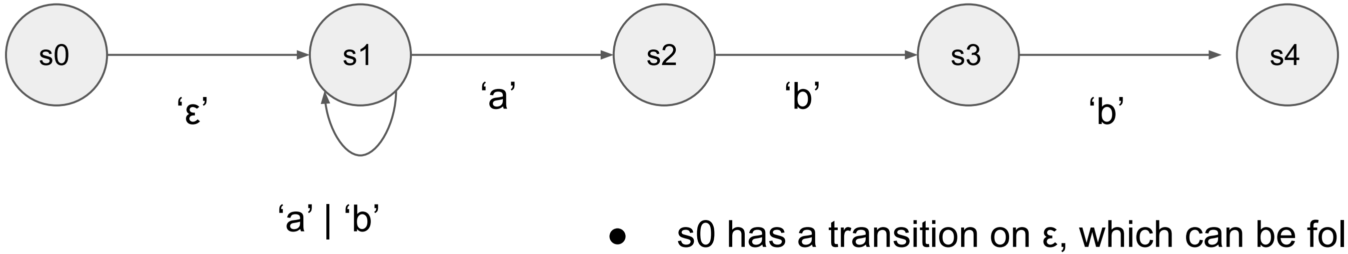
Skeleton recogniser

```
c = next_character()

state = “s0”
while c := EOF:
    state = δ(s, c)
    c = next_character()

if (state final):
  return success
else:
  return error
```

Non-Determinism

Deterministics 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 ?

- s0 has a transition on ε, 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 ε transition

Non-deterministic finite state automata (NFA):

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

This means we ***might have to backtrack***

Automatic Lexer Generation

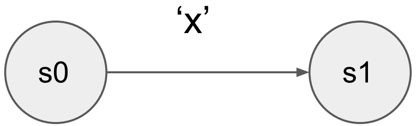
It is possible to systematically generate a lexer for any regular expression.

This can be done in three steps:

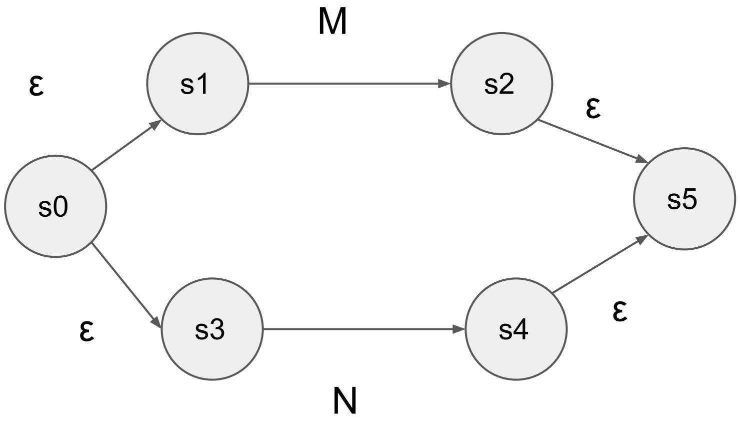
1. regular expression (RE) → non-deterministic finite automata (NFA)
2. NFA → deterministic finite automata (DFA)
3. DFA → generated lexer

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

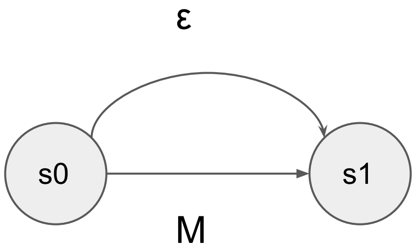
‘x’



M | N

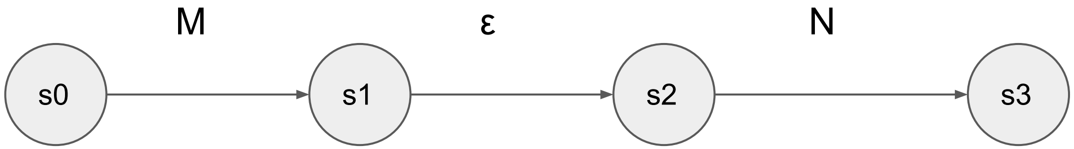


[M]

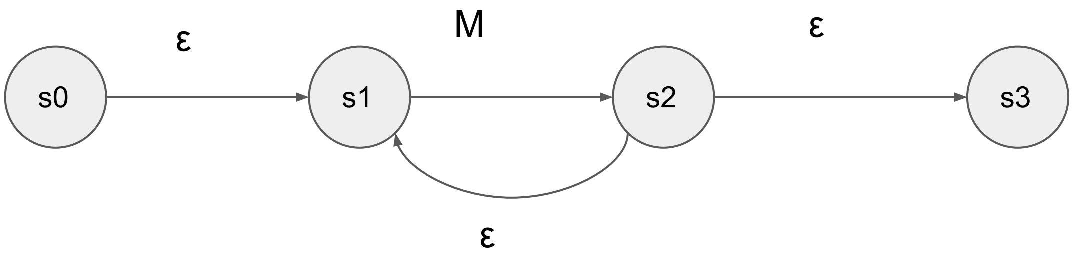


1st step: RE → NFA (Ken Thompson, CACM, 1968)

M N



M+



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 si and the states of the DFA we are building are labelled qi.

We have two key functions:

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

Algorithm

The Subset Construction algorithm (Fixed point iteration)

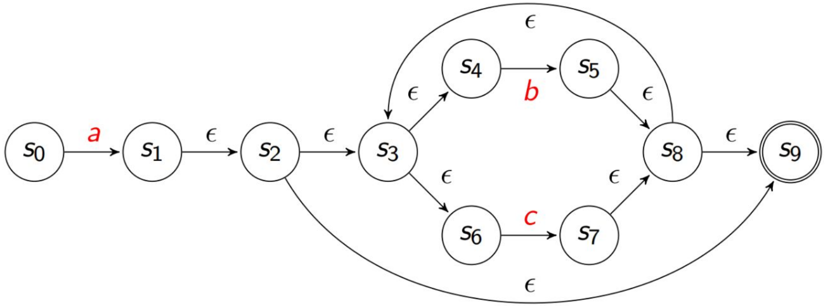
$q_0 = \epsilon\text{-}closure(s_0)$; $Q = \{q_0\}$; add q_0 to WorkList
while (WorkList not empty)
  remove q from WorkList
  for each $\alpha \in \Sigma$
    $subset = \epsilon\text{-}closure(reachable(q, \alpha))$
    $\delta(q, \alpha) = subset$
    if ($subset \notin Q$) then
      add $subset$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

The algorithm (in English)

- Start from start state s_0 of the NFA, compute its 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

NFA for a(b|c)*

$a(b|c)^*$



		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

DFA for a(b|c)*

Graph

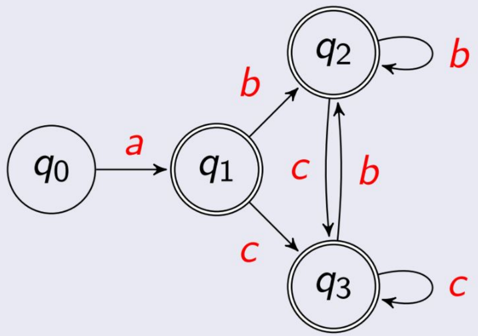


Table encoding

	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

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
- Recursive descent parser