NTNU | Norwegian University of Science and Technology

Compiler Construction

Lecture 3: Scanner Generators

Michael Engel

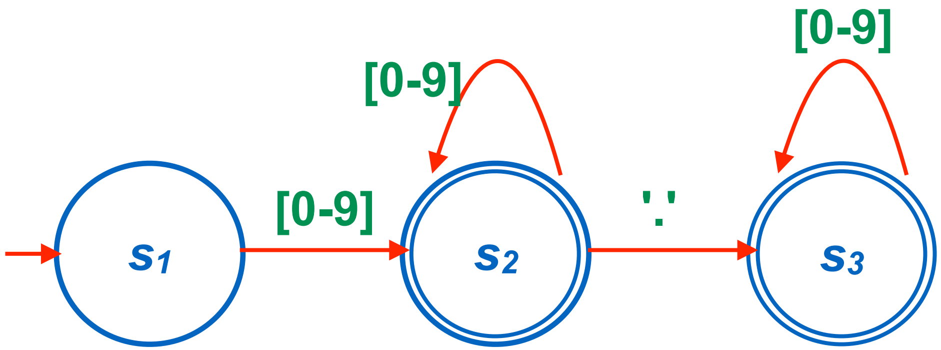
Includes material by Jan Christian Meyer

Overview

- DFAs and regular expressions
- Nondeterministic finite automata (NFA)
- From regular expressions to NFAs

The DFA, again

This DFA from the previous lecture…



…was able to tell you whether a character sequence is a valid decimal number (integer + optional fractional part) or not

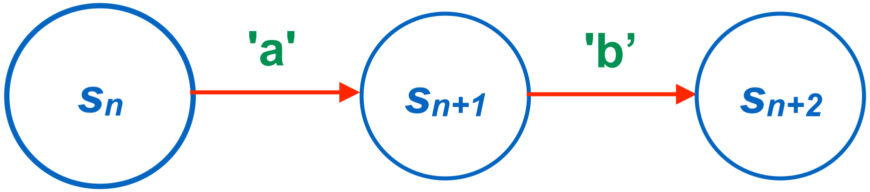
- Start with the initial state s_1, then follow the edges

More about lexemes

- ***Lexeme***
  - Lexemes are units of lexical analysis, words
  - Like dictionary entries

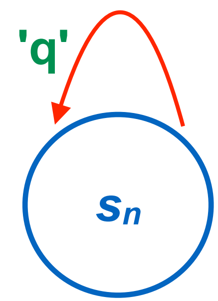
Common patterns in lexemes

- ***Sequences*** of specific parts
  - chains of states in the graph



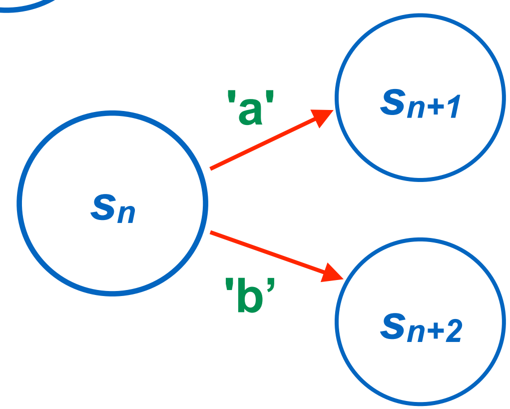
Sequence “ab”

- ***Repetition***
  - loops in the graph



Any number (>=0) of 'q’s

- ***Alternatives***
  - different paths in the graph



Either 'a' or 'b'

DFA formal notation

Formal definition: DFA = 5-tuple (Q, Σ, δ, q_0, F)

Q is a finite set called the ***states***,

Σ is a finite set called the ***alphabet***,

δ: Q×Σ $\rightarrow$ ***Q*** is the ***transition function***,

q_0 $\in$ ***Q*** is the ***start state***, and

F $\subseteq$ ***Q*** is the set of ***accepting states***

$Q = \{s_1, s_2, s_3\}$

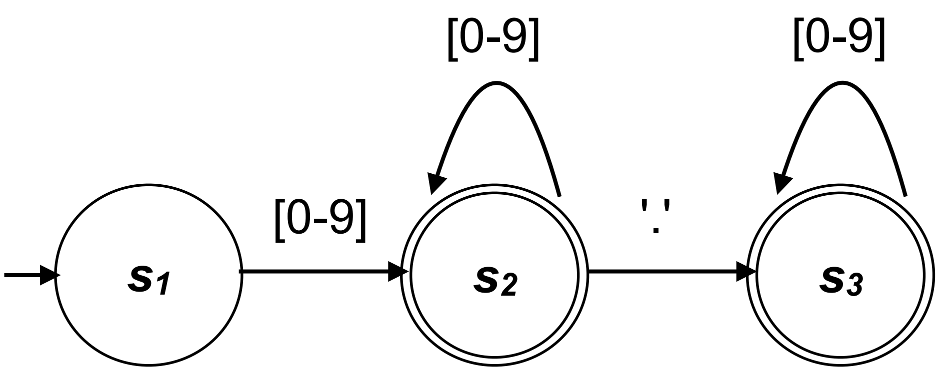
$\Sigma = \{0,1,2,3,4,5,6,7,8,9,.\}$

$q_0 = s_1$

$F = \{s_2, s_3\}$

δ =

δ	0	1	2	3	4	5	6	7	8	9	.
s_1	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	er
s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_2	s_3
s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	s_3	er



Alphabets in DFAs

- ***Alphabet***: finite set of symbols (characters)
  - {0,1} is the alphabet of binary strings
  - [A-Za-z0-9] is the alphabet of alphanumeric strings

- A ***language*** is a set of valid strings (sequences of symbols) over an alphabet
  - L = {000, 010, 100, 110} is the language of "even, positive binary numbers less than 8"

- A finite automaton ***accepts a language***
  - it decides whether or not a given string belongs to the language described by it

Operations on languages

- ***Union*** of languages: $s \in L_1 \cup L_2$ if $s \in L_1$ or $s \in L_2$
- ***Concatenation***: $L_1 L_2 = \{ s_1 s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Concatenation of a language with itself: "multiplication" (***Cartesian product***):
  $LLL = \{ s_1 s_2 s_3 \mid s_1 \in L \text{ and } s_2 \in L \text{ and } s_3 \in L \}$
- ***Closures***
  - $L^* = \cup_{i=\mathbf{0},\mathbf{1},\mathbf{2},\ldots} L^i$: "Kleene closure": **0** or more strings from L
  - $L^+ = \cup_{i=\mathbf{1},\mathbf{2},\ldots} L^i$: "Positive closure": **1** or more strings from L

Operations on languages: examples

- ***Union*** of languages: $s \in L_1 \cup L_2$ if $s \in L_1$ or $s \in L_2$
  - L_1 = {000, 010, 100, 110}, L_2 = {001, 011, 101, 111}
    $\Rightarrow L_1 \cup L_2$ = {000, 001, 010, 011, 100, 101, 110, 111}
- ***Concatenation**:* $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
  - L_1 = {“ab”, “c”}, L_2 = {“x”}
    $\Rightarrow L_1L_2$ = {“abx”, “cx”}
- Concatenation of a language with itself: “multiplication” (***Cartesian product***):
  $LLL = \{ s_1s_2s_3 \mid s_1 \in L \text{ and } s_2 \in L \text{ and } s_3 \in L \}$
  - L = {“a”, “b”}

  $\Rightarrow LLL$ =
  { “aaa”, “aab”, “aba”, “abb”, “baa”, “bab”, “bba”, “bbb" }

Operations on languages: examples

- ***Closures***
  - $L^* = \cup_{i=\mathbf{0},\mathbf{1},\mathbf{2},\ldots} L^i$: “Kleene closure”: **0** or more strings from L

    0 strings = **empty word** ε (“epsilon”)

    {"ab","c"}* = { **ε**, "ab", "c", "abab", "abc", "cab", "cc", "ababab", "ababc", "abcab", "abcc", "cabab", "cabc", "ccab", "ccc", ...}

  - $L^+ = \cup_{i=\mathbf{1},\mathbf{2},\ldots} L^i$: “Positive closure”: **1** or more strings from *L*

    {"a", "b", “c”}⁺ = { "a", "b", "c", "aa", "ab", "ac", "ba", "bb", "bc", "ca", "cb", "cc", "aaa", "aab", …}

  - $L^* = \{\varepsilon\} \cup L^+$

Regular expressions (“regexp”)

Given: *Empty string* ε (epsilon), Alphabet **Σ** (sigma)

Recursive definition of regular expressions:

Basis

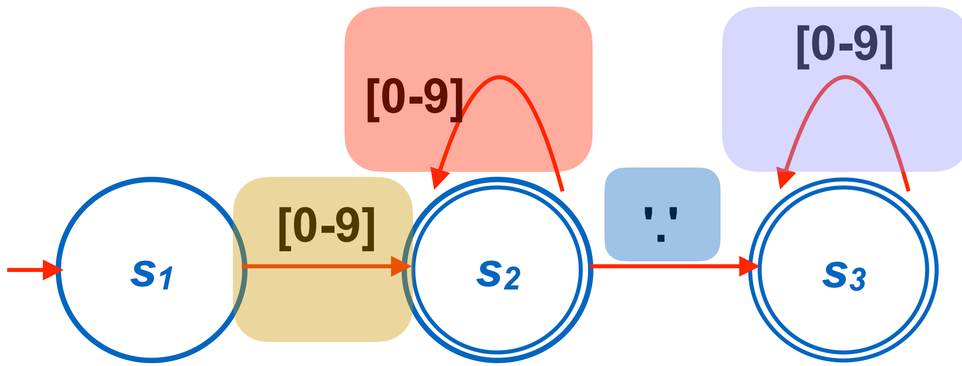
- ε is a regular expression, *L*(ε) is the language with only ε in it
- If a is in Σ, then a is also a regular expression, *L*(a) is the language with only a in it

Induction

- If r_1 and r_2 are regexps ⇒ $r_1 \mid r_2$ is regexp for $L(r_1) \cup L(r_2)$ (***selection***)
- If r_1 and r_2 are regexps ⇒ $r_1 r_2$ is regexp for $L(r_1)L(r_2)$ (***concatenation***)
- If r is a regular expression ⇒ r* denotes L(r)* (***Kleene closure***)
- (r) is a regular expression denoting L(r)
  (***We can add parentheses to group parts of the regexp***)

DFAs and regular expressions

Again, the DFA which accepts decimal numbers:



This DFA corresponds to the following regular expression:

[0-9] [0-9]* (. [0-9]*)?

optional, since state s_2 accepts

This dot "." here stands for the character "." (ASCII 0x2E), not for any arbitrary character!

Abbreviated notation used for regexps:

. – any character ∈ **Σ**

[abc] – either 'a' or 'b' or 'c'

[a-d] – characters from 'a' to 'd' inclusive

? – either zero or one repetition

Three ways to describe a language

- Graphs
  - provide a quick overview of the structure
- Tables
  - help writing programs to implement the DFA
- Regular expressions
  - help generating accepting automata automatically

Regular languages

- All three representations are equivalent
  - We have not shown a formal way to transform one representations into the other and did not prove this
  - Maybe you can still see it?

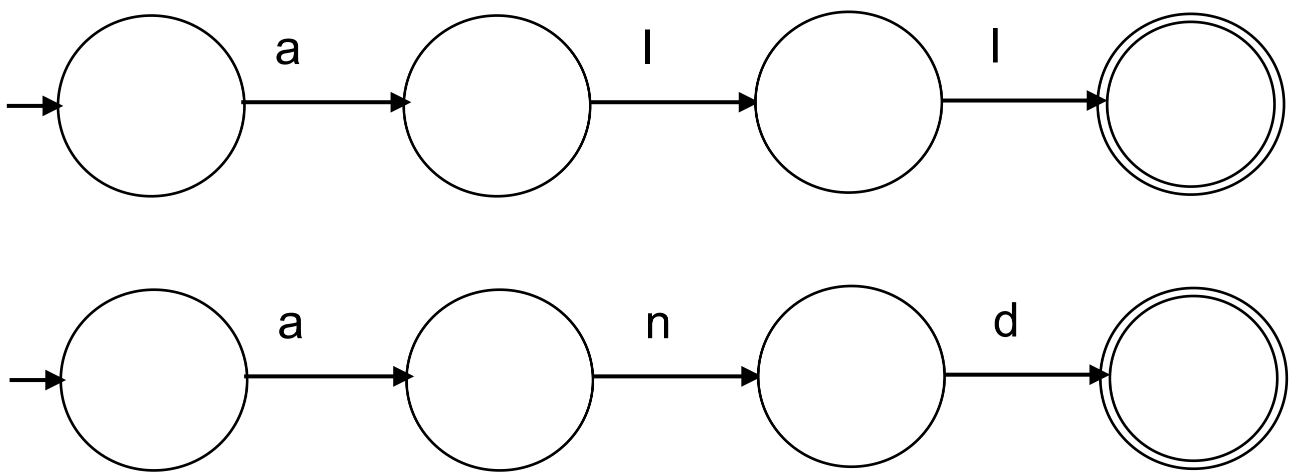
- The ***family*** of languages that can be recognized by automata/regexps is called ***regular languages***

- They are an important and powerful class of languages
  - However, they do not cover all use cases
  - e.g., *recursion* cannot be specified using regexps
  - more on this later…

Combining automata

Wanted: language that includes the words {“all”, “and”}

- Simple DFAs to detect each of the words separately:

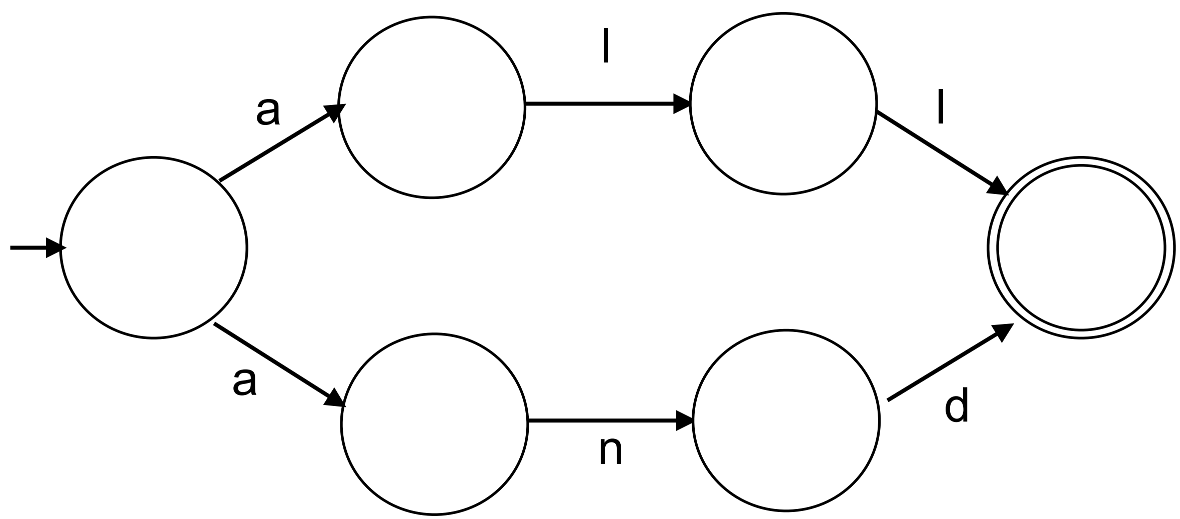


We omit the numbering of states if the specific number is not relevant for an example

Combining automata

Wanted: language that includes the words {“all”, “and”}

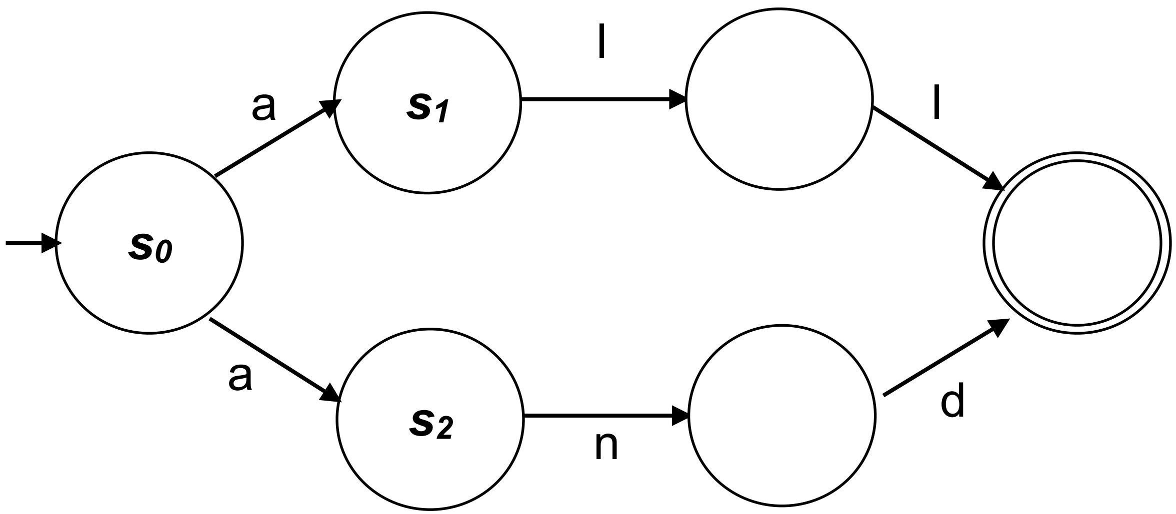
- Can we build an automaton to detect ***both*** words?
  - How about combining both DFAs?
  - Simply join the starting and accepting states of both:



Now we have a (small) problem

"Walking" the DFA does not work any more

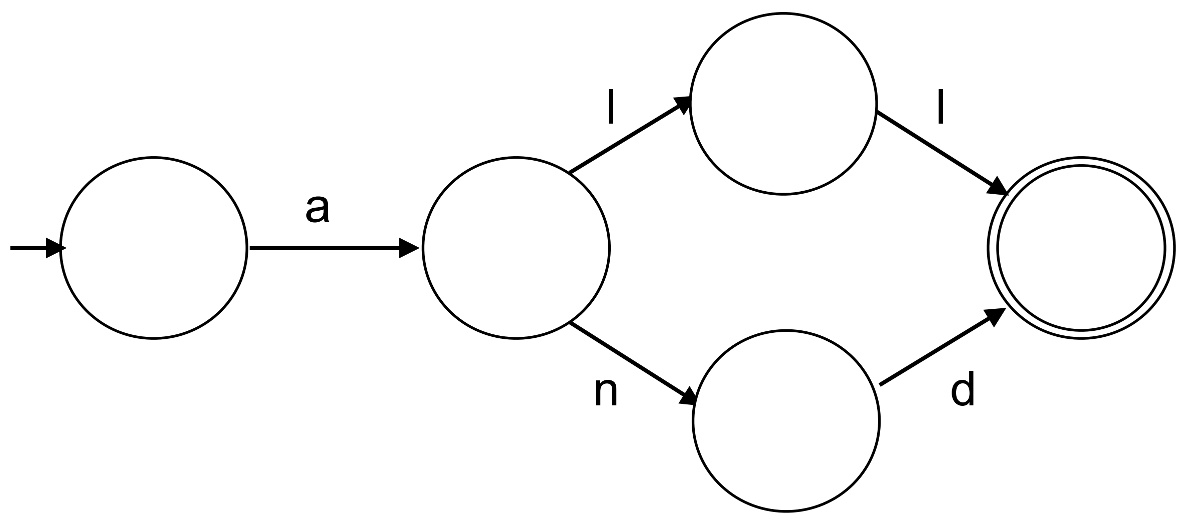
- Starting at s_0 and reading 'a', the next state can be s_1 or s_2
- If we read an 'a', chose s_1 and then read an 'n' ⇒ wrong path
- We would need to go to states s_1 and s_2 ***at the same time***
  - Otherwise, we would need some way to backtrack to s_0



An obvious solution

Combine states states s_1 and s_2
⇒ postpone the decision which path to choose

- Walking the DFA works again!
- Need to determine which parts both words have in common *(can that be generalized?)*

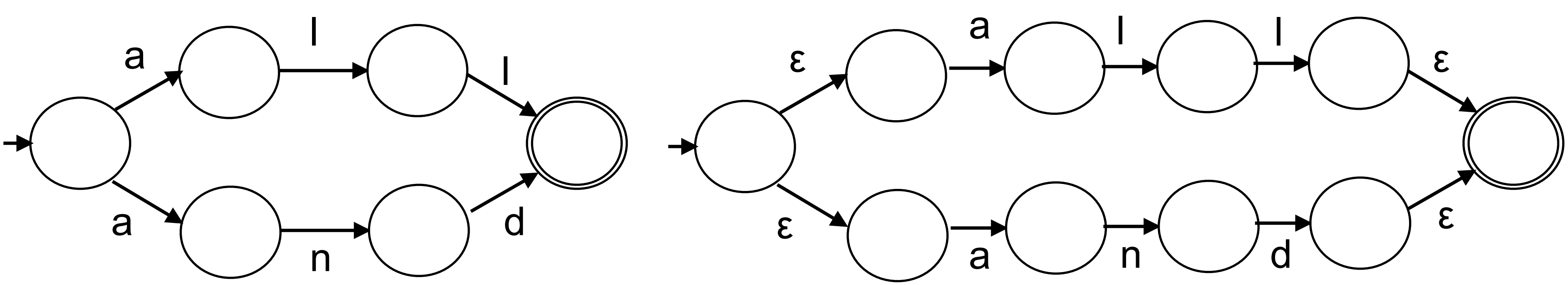


Non-Deterministic Finite Automata

Idea:
admit multiple transitions from one state on the same character

- Alternative: allow transitions on the empty input ε
  (i.e., without reading a character)

- Both notations are equivalent:



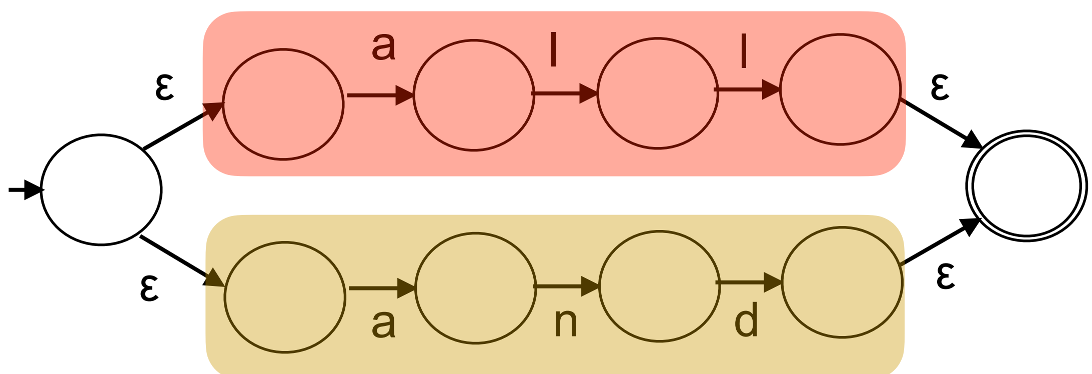
NFAs and regular expressions

NFAs can easily be constructed from regular expressions

- For our example, the regexp would be: { all | and }
  (equivalent deterministic variant: a{ll | nd})

- The two sub-automata can easily be identified in the graph:

sub-automaton ("machine") 1



sub-automaton ("machine") 2

Constructing a scanner

What are the parts of a regexp again?

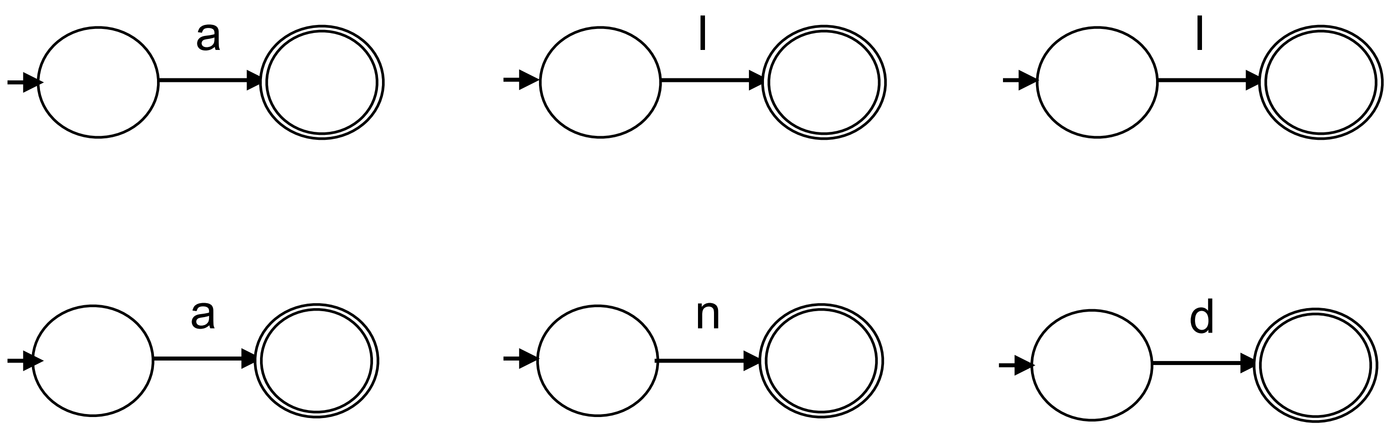
1. a (single) character: stands for itself (or ε – that's not shown)
2. concatenation: R_1R_2
3. selection: $R_1 \mid R_2$
4. grouping: (R_1)
5. Kleene closure: R_1^*

- We can construct an NFA for each of these
  …as long as R_1 and R_2 are regexps (⇒ recursive definition)
  - Note: each DFA is also an NFA (with zero ε-transitions)
  - Formal: the set of DFAs is a subset of the set of NFAs

Constructing a scanner: characters

Single characters (and epsilons) in a regexp become transitions between two states in an NFA

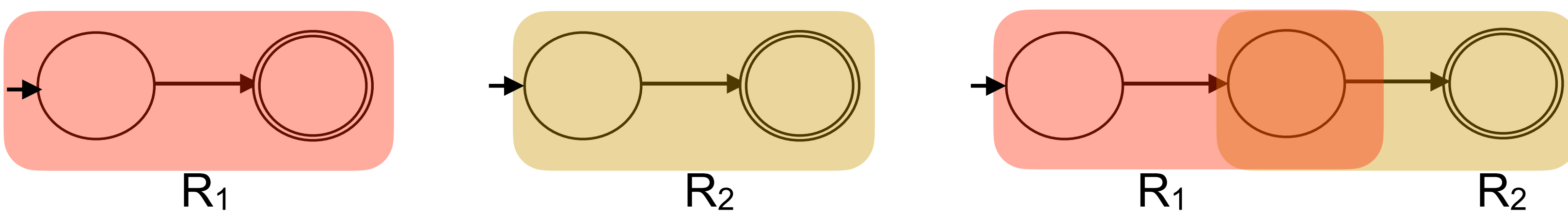
- For our example { all | and }, the transitions are thus:



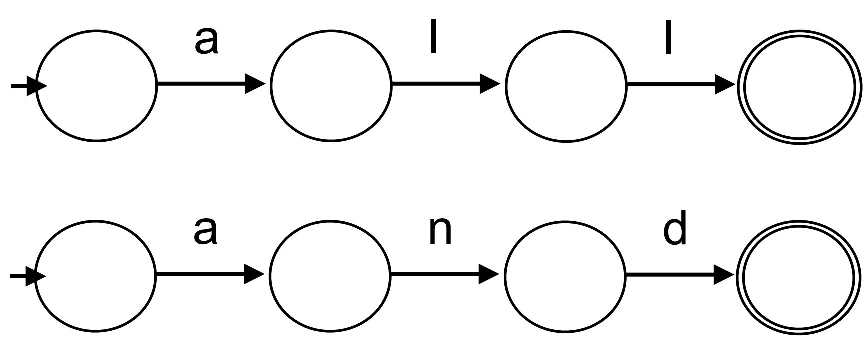
Now we can combine these simple regexps…

Constructing a scanner: concatenation

Where R_1R_2 are concatenated, join the accepting state of R_1 with the start state of R_2

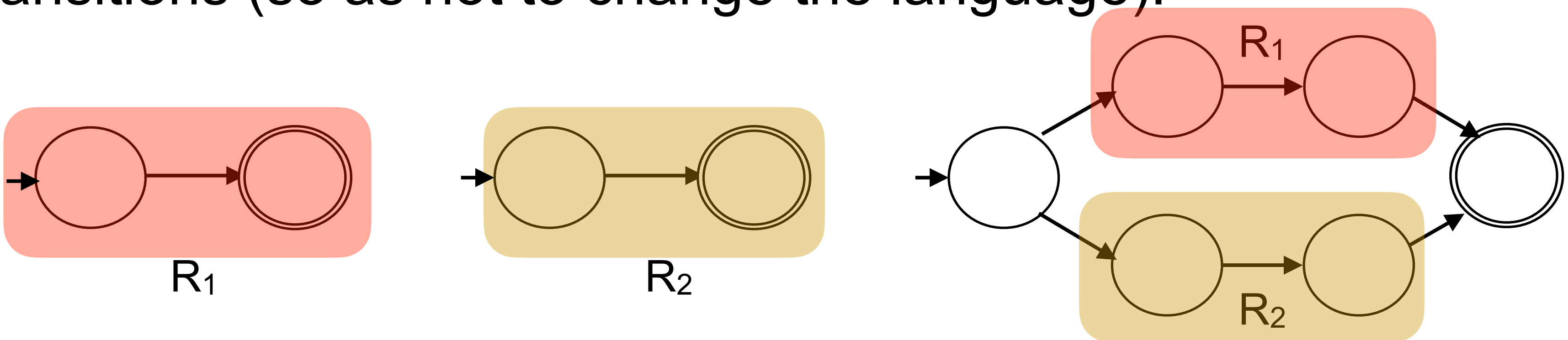


- In our example:

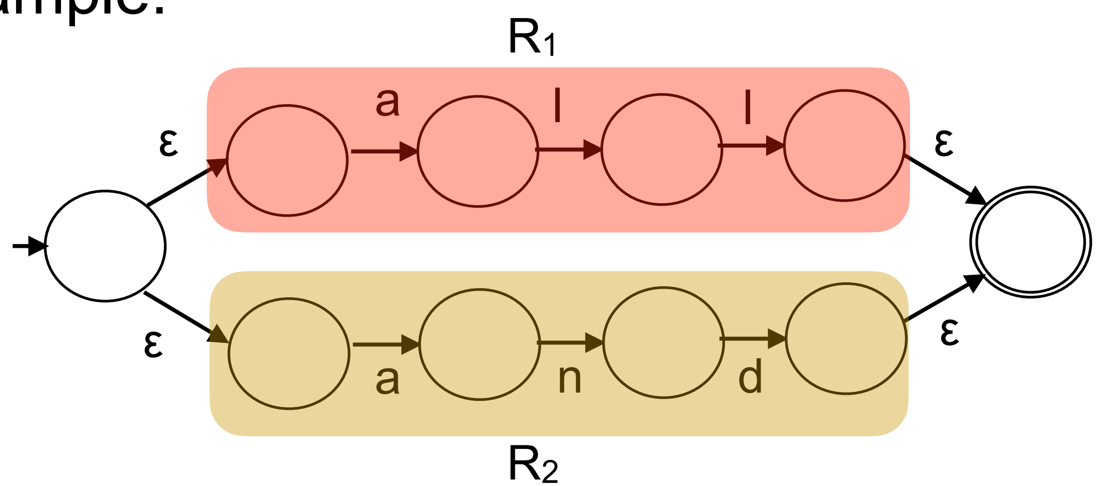


Constructing a scanner: selection

Introduce new start and accept states, attach them using ε-transitions (so as not to change the language):



- In our example:



Constructing a scanner: grouping

Parentheses just delimit which parts of an expression to treat as a (sub-)automaton

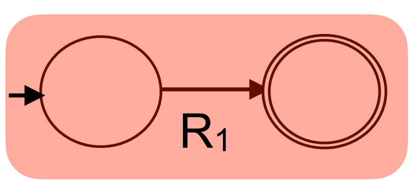
- they appear in the form of its structure, but not as nodes or edges

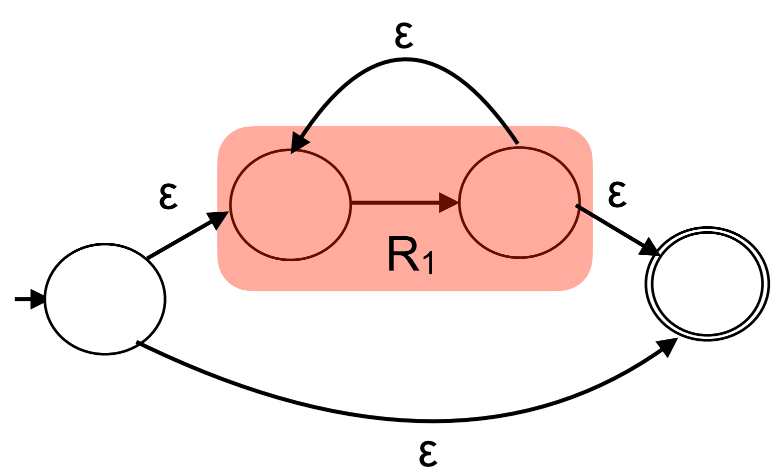
In our example, the automaton for (all | and) is identical to the one for ((a) (l) (l) | (a) (n) (d))

Constructing a scanner: Kleene clos.

R_1* means zero or more concatenations of R_1

- Introduce new start and accept states and add ε-transitions to
  - Accept a single walk through R_1
  - Loop back to the start of R_1 to allow any number of repetitions
  - Bypass R_1 entirely (zero walkthroughs, i.e. R_1 does not occur)





What have we achieved so far?

- We have shown (by construction) that we can construct an NFA for <u>any</u> regular expression
  - independent of the contents of that expression
- This is called the *McNaughton-Thompson-Yamada algorithm* [1][2]

- But what about the positive closure, R_1^+?
  - It can be made from concatenation and Kleene closure, try it yourself
  - It's handy to have as notation, but not necessary to prove what we wanted here

Some wise words and references

Jamie> Some people, when confronted with a problem, think "I know,
Jamie> I'll use regular expressions." Now they have two problems.

Jamie Zawinksi, early Netscape engineer
in a 1997 Usenet article
<33F0C496.370D7C45@netscape.com>

[1] R. McNaughton, H. Yamada (Mar 1960):
"Regular Expressions and State Graphs for Automata".
IEEE Trans. on Electronic Computers. 9 (1): 39–47. doi:10.1109/TEC.1960.5221603

[2] Ken Thompson (Jun 1968):
"Programming Techniques: Regular expression search algorithm".
Communications of the ACM. 11 (6): 419–422. doi:10.1145/363347.363387