Lexical Analysis

Jim Royer

CIS 352

January 31, 2019

https://xkcd.com/208/

References

The following is partly based on

Basics of Compiler Design
by Torben Mogensen
http://hjemmesider.diku.dk/~torbenm/Basics/

Regular Expressions and Automata using Haskell
Simon Thompson
http://www.haskellcraft.com/craft3e/Reg_exps.html

➤ The Syntactic Side of Languages

Natural Languages

stream of phonemes $\xrightarrow[\text{analysis}]{\text{via lexical}}$ stream of words $\xrightarrow{\text{via parsing}}$ sentences

Artificial Languages

stream of characters $\xrightarrow[\text{analysis}]{\text{via lexical}}$ stream of tokens $\xrightarrow{\text{via parsing}}$ abstract syntax

What is a token?

Variable names, numerals, operators (e.g., $+$, $/$, etc.), key-words, ...

Lexical structure is typically specified via *regular expressions*.

➤ Regular Expressions *(S. Kleene, 1951, 1956)*

Definition 1.

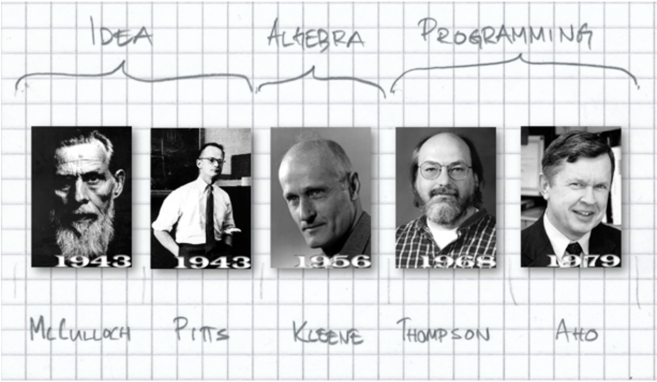
A regular expression has one of six forms:

$\emptyset$	—	matches no string[§]
ϵ	—	matches the empty string
`x`	—	matches the character '`x`'
$(r_1 \vert r_2)$	—	matches the strings matched by r_1 or r_2
$(r_1 r_2)$	—	matches the strings $w_1 w_2$ where w_1 matches r_1 and w_2 matches r_2
$(r)^*$	—	matches ϵ and the strings $w_1 \dots w_k$ where $k > 0$ and each w_i matches r

We omit the parens in $(r_1|r_2)$, $(r_1 r_2)$, and $(r)^*$ when we can.

[§]Both Thompson and Mogensen omit this form, and henceforth, so shall we.
($\emptyset$ is necessary for algebraic treatments of regular languages.)

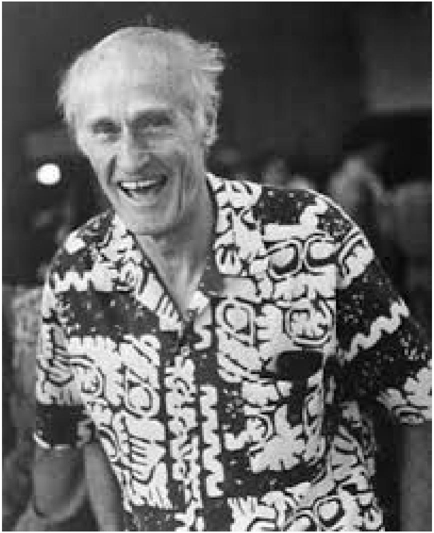
Regular Expressions, History



Picture from: https://blog.staffannoteberg.com/2013/01/30/regular-expressions-a-brief-history/

- M&P: math model for neurons
- K: Extended M&P, reg. exps.
- T&A: grep, REGEX library

Steve Kleene, Haskell Curry & Bruce Kleene



Regular Expressions: Examples

1. Sheep Language = $\{ ba!, baa!, baaa!, baaaa!, \ldots \}$.
   $baa^*! = (b((a(a^*))!))$ matches exactly the Sheep Language strings.
2. $(0|1)^*$ matches exactly the strings over 0 and 1, including ϵ.
3. $(\epsilon|(1(0|1)^*))1$ matches exactly the binary representation of odd integers.
   —*more examples shortly*—

Notation

$r \Downarrow s \;\equiv_{\text{def}}$ regular expression r matches string s.

Big-Step Rules for RegEx Matching

ϵ-match: $\dfrac{}{\epsilon \Downarrow \epsilon}$ *Literal-match:* $\dfrac{}{x \Downarrow x}$

|-match₁: $\dfrac{r_1 \Downarrow s}{(r_1|r_2) \Downarrow s}$ *|-match₂:* $\dfrac{r_2 \Downarrow s}{(r_1|r_2) \Downarrow s}$

++-match: $\dfrac{r_1 \Downarrow s_1 \quad r_2 \Downarrow s_2}{(r_1 r_2) \Downarrow s} \; (s = s_1 {+\!+} s_2)$

**-match₁:* $\dfrac{}{r^* \Downarrow \epsilon}$ **-match₂:* $\dfrac{r \Downarrow s_1 \quad r^* \Downarrow s_2}{r^* \Downarrow s} \; (s = s_1 {+\!+} s_2)$

[**Stage direction:** Copy these onto the board, but leave some room.]

Applying the Big-Step Rules

$$
\text{++}\;\dfrac{
  \text{Lit}\;\dfrac{}{b \Downarrow b}
  \qquad
  \text{++}\;\dfrac{
    \text{++}\;\dfrac{
      \text{Lit}\;\dfrac{}{a \Downarrow a}
      \qquad
      {*_2}\;\dfrac{
        \text{Lit}\;\dfrac{}{a \Downarrow a}
        \qquad
        {*_2}\;\dfrac{
          \text{Lit}\;\dfrac{}{a \Downarrow a}
          \qquad
          {*_1}\;\dfrac{}{a^* \Downarrow \epsilon}
        }{a^* \Downarrow a}
      }{a^* \Downarrow aa}
    }{(aa^*) \Downarrow aaa}
    \qquad
    \text{Lit}\;\dfrac{}{! \Downarrow !}
  }{((aa^*)!) \Downarrow aaa!}
}{b((aa^*)!) \Downarrow baaa!}
$$

Class Exercise. Work out derivations for:

1. $(0|1)^* \Downarrow 0101$
2. $(0|1)^*((01)|(10)) \Downarrow 0110$

Applying the Big-Step Rules

$$\text{++}\ \dfrac{\text{Lit}\ \dfrac{}{b \Downarrow b} \qquad \text{++}\ \dfrac{\text{++}\ \dfrac{\text{Lit}\ \dfrac{}{a \Downarrow a} \qquad {*_2}\ \dfrac{\text{Lit}\ \dfrac{}{a \Downarrow a} \qquad {*_2}\ \dfrac{\text{Lit}\ \dfrac{}{a \Downarrow a} \qquad {*_1}\ \dfrac{}{a^* \Downarrow \epsilon}}{a^* \Downarrow a}}{a^* \Downarrow aa}}{(aa^*) \Downarrow aaa} \qquad \text{Lit}\ \dfrac{}{! \Downarrow !}}{((aa^*)!) \Downarrow aaa!}}{b((aa^*)!) \Downarrow baaa!}$$

Class Exercise. Work out derivations for:

1. $(0|1)^* \Downarrow 0101$
2. $(0|1)^*((01)|(10)) \Downarrow 0110$

Class Exercise 1

$${*_2}\ \dfrac{|_1\ \dfrac{\text{Lit}\ \dfrac{}{0 \Downarrow 0}}{(0|1) \Downarrow 0} \qquad {*_2}\ \dfrac{|_2\ \dfrac{\text{Lit}\ \dfrac{}{1 \Downarrow 1}}{(0|1) \Downarrow 1} \qquad {*_2}\ \dfrac{|_1\ \dfrac{\text{Lit}\ \dfrac{}{0 \Downarrow 0}}{(0|1) \Downarrow 0} \qquad {*_2}\ \dfrac{|_2\ \dfrac{\text{Lit}\ \dfrac{}{1 \Downarrow 1}}{(0|1) \Downarrow 1} \qquad {*_1}\ \dfrac{}{(0|1)^* \Downarrow \epsilon}}{(0|1)^* \Downarrow 1}\ \text{"1"="1"++}\epsilon}{(0|1)^* \Downarrow 01}\ \text{"01"="0"++"1"}}{(0|1)^* \Downarrow 101}\ \text{"101"="1"++"01"}}{(0|1)^* \Downarrow 0101}\ \text{"0101"="0"++"101"}$$

Class Exercise 2

$$\text{++}\ \dfrac{{*_2}\ \dfrac{|_1\ \dfrac{\text{Lit}\ \dfrac{}{0 \Downarrow 0}}{(0|1) \Downarrow 0} \qquad {*_2}\ \dfrac{|_2\ \dfrac{\text{Lit}\ \dfrac{}{1 \Downarrow 1}}{(0|1) \Downarrow 1} \qquad {*_1}\ \dfrac{}{(0|1)^* \Downarrow \epsilon}}{(0|1)^* \Downarrow 1}}{(0|1)^* \Downarrow 01} \qquad |_2\ \dfrac{\text{++}\ \dfrac{\text{Lit}\ \dfrac{}{1 \Downarrow 1} \qquad \text{Lit}\ \dfrac{}{0 \Downarrow 0}}{10 \Downarrow 10}}{((01)|(10)) \Downarrow 10}}{(0|1)^*((01)|(10)) \Downarrow 0110}$$

Matching Regular Expressions in Haskell, I

Abstract Syntax

```
data Reg = Epsilon
         | Literal Char
         | Or Reg Reg
         | Then Reg Reg
         | Star Reg
           deriving (Eq)
```

```
matches ::  Reg -> String -> Bool

matches Epsilon st
  = (st == "")
matches (Literal ch) st
  = (st == [ch])
matches (Or r1 r2) st
  = matches r1 st || matches r2 st
  ⋮ (continued)
```

Credits/Pointers

- The code here is based on work by Simon Thompson.
  See: `http://www.haskellcraft.com/craft3e/Reg_exps.html`
- Also see `RegExp.hs` and `Matches.hs` in
  `http://www.cis.syr.edu/courses/cis352/code/RegExp/`
  our local copy of Thompson's code.

Matching Regular Expressions in Haskell, I

Abstract Syntax

```
data Reg = Epsilon
         | Literal Char
         | Or Reg Reg
         | Then Reg Reg
         | Star Reg
           deriving (Eq)
```

```
matches ::  Reg -> String -> Bool
matches Epsilon st
  = (st == "")
matches (Literal ch) st
  = (st == [ch])
matches (Or r1 r2) st
  = matches r1 st || matches r2 st
  ⋮
```

(continued)

Credits/Pointers

- The code here is based on work by Simon Thompson.
  See: http://www.haskellcraft.com/craft3e/Reg_exps.html
- Also see `RegExp.hs` and `Matches.hs` in
  http://www.cis.syr.edu/courses/cis352/code/RegExp/
  our local copy of Thompson's code.

Stage Directions:

Open up

- `RegExp.hs`
- `Matches.hs`

Matching Regular Expressions in Haskell, II

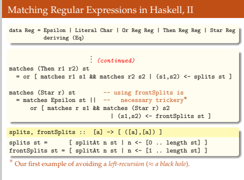
```
data Reg = Epsilon | Literal Char | Or Reg Reg | Then Reg Reg | Star Reg
           deriving (Eq)
```

```
                          ⋮ (continued)
matches (Then r1 r2) st
  = or [ matches r1 s1 && matches r2 s2 | (s1,s2) <- splits st ]

matches (Star r) st           -- using frontSplits is
  = matches Epsilon st ||     --   necessary trickery*
      or [ matches r s1 && matches (Star r) s2
                              | (s1,s2) <- frontSplits st ]
```

```
splits, frontSplits ::  [a] -> [ ([a],[a]) ]
splits st =       [ splitAt n st | n <- [0 .. length st] ]
frontSplits st = [ splitAt n st | n <- [1 .. length st] ]
```

* Our first example of avoiding a *left-recursion* (≈ *a black hole*).



Stage Directions:

Replace the case for `Star` with

```
matches (Star r) st
   = matches Epsilon st ||
     or [ matches r s1 && matches (Star r) s2
                       | (s1,s2) <- splits st ]
```

and

```
(Star (Epsilon `Or` (Literal 'a'))) "aa"
```

try

Definition 2.

Suppose r is a regular expression and A and B are sets of strings.

a. $L(r)$ = the set of strings matched by r.
b. $A \cdot B = \{\, w_a w_b \mid w_a \in A,\ w_b \in B \,\}$.
c. $A^0 = \{\, \epsilon \,\},\ \ A^1 = A,\ \ A^2 = A \cdot A,\ \ A^3 = A \cdot A \cdot A,\ \ \ldots$

Thus:

$$
\begin{array}{rcl}
L(\epsilon) & = & \{\, \epsilon \,\} \\
L(\mathtt{x}) & = & \{\, \mathtt{x} \,\} \\
L(r_1 | r_2) & = & L(r_1) \cup L(r_2) \\
L(r_1 r_2) & = & L(r_1) \cdot L(r_2) \\
L(r^*) & = & \{\, \epsilon \,\} \cup L(r) \cdot L(r^*) \quad = \quad \bigcup_{i \geq 0} L(r)^i
\end{array}
$$

Short Cuts (Mogensen, §2.1.1)

- We can write $(0|1|2|3|4|5|6|7|8|9)$ as $[0123456789]$ or $[0-9]$.
- $r^+ = r\,r^*$, i.e.,

$$\begin{array}{rcl} r^* & \equiv & \text{0 more more matches of } r \\ r^+ & \equiv & \text{1 more more matches of } r \end{array}$$

- $r? = r|\epsilon \quad \equiv \quad$ 0 or 1 matches of r.

Examples

- $[a-zA-Z]$ = all alphabetic characters
- $(0|([1-9][0-9]^*))$ = all natural number constants
- $[a-zA-Z\_][a-zA-Z0-9]^* \equiv$ C variable names
- "$\big([a-zA-Z0-9]\big|\backslash[a-zA-Z0-9]\big)^*$" $\equiv$ C string constants

Regular Expressions with Their Work Boots On

- grep, egrep, fgrep — print lines matching a pattern
  See `http://en.wikipedia.org/wiki/Grep`
- Also see `tr`, `sed`, ...
  (The original Unix developers knew their automata theory cold.)
- See `http://perldoc.perl.org/perlre.html`.
  (Folks in bioinformatics know their pattern matching cold.)
- See `https://en.wikipedia.org/wiki/Comparison_of_regular_expression_engines`.

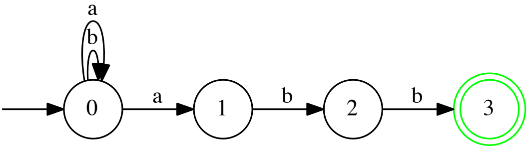
A *Non-deterministic Finite Automaton* (abbreviated *NFA*) consists of:

- A finite set of states, S.
- A finite set of moves *(labeled edges between states)*
  (Moves are labeled by either ϵ or a $c \in \Sigma =$ the input alphabet)
- A start state (in S).
- A set of terminal or final states (a subset of S).

Example 3.

$S = \{0, 1, 2, 3\}$, start state = 0, final sets = $\{3\}$

moves = $\{0 \xrightarrow{a} 0,\ 0 \xrightarrow{b} 0,\ 0 \xrightarrow{a} 1,\ 1 \xrightarrow{b} 2,\ 2 \xrightarrow{b} 3\}$



The Data.Set Module

To implement NFA's we need a module for representing sets.
We use:
http://hackage.haskell.org/packages/archive/containers/latest/doc/html/Data-Set.html

Data.Set

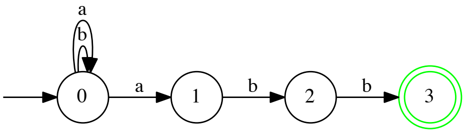
```
         empty :: Set a
      fromList :: Ord a => [a] -> Set a
  intersection :: Ord a => Set a -> Set a -> Set a
  Data.Set.map :: (Ord a, Ord b) => (a -> b) -> Set a -> Set b
     singleton :: a -> Set a
          size :: Set a -> Int
        toList :: Set a -> [a]
         union :: Ord a => Set a -> Set a -> Set a
               etc.
```

NFAs represented in Haskell

```
data Move a = Move a Char a | Emove a a
              deriving (Eq,Ord,Show)

data Nfa a  = NFA (Set a) (Set (Move a)) a (Set a)
              deriving (Eq,Show)
```

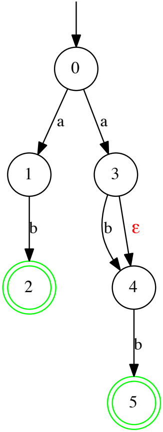
```
machM ::  Nfa Int
machM = NFA
        (S.fromList [0..3])
        (S.fromList [Move 0 'a' 0,  Move 0 'a' 1,  Move 0 'b' 0,
                     Move 1 'b' 2,  Move 2 'b' 3] )
        0
        (S.singleton 3)
```



Another Example NFA

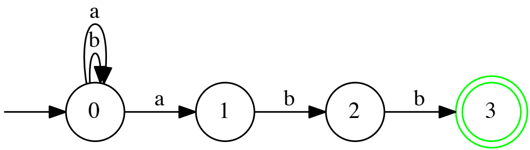
```
machN = NFA
        (S.fromList [0..5])
        (S.fromList [Move 0 'a' 1,
                     Move 1 'b' 2,
                     Move 0 'a' 3,
                     Move 3 'b' 4,
                     Emove 3 4,
                     Move 4 'b' 5])
        0
        (S.fromList [2,5])
```

Note the two sorts of nondeterminism this machine exhibits.

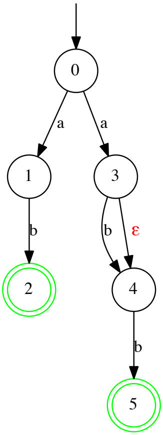


Accepting and rejecting strings

- What is the accepting path of *abb* through *M*?
- What other paths are possible?
- What are the accepting paths of *ab* through *N*?
- What happens with *N* and *aa*?



Machine *M*



Machine *N*

A small-step semantics for an NFA

Notation

For $M = (States, Moves, start, Final)$:

- $M \vdash s \overset{a}{\Longrightarrow} s' \;\equiv_{\text{def}}\; (s, a, s') \in Moves.$
- $M \vdash s \overset{\epsilon}{\Longrightarrow} s' \;\equiv_{\text{def}}\; (s, \epsilon, s') \in Moves.$

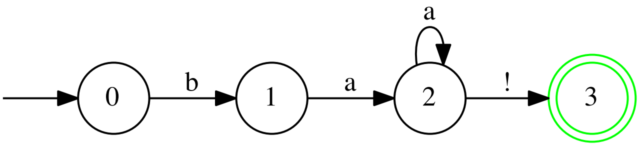
$$\frac{}{M \vdash s \overset{a}{\Longrightarrow} s'} \; \big((s, a, s') \in Moves\big)$$

$$\frac{}{M \vdash s \overset{\epsilon}{\Longrightarrow} s'} \; \big((s, \epsilon, s') \in Moves\big)$$

[**Stage direction:** Copy these onto the board.]

Applying the Small-Step Rules, 1

$$M = (\{\,0,1,2,3\,\}, \{\,0 \xrightarrow{b} 1,\ 1 \xrightarrow{a} 2,\ 2 \xrightarrow{a} 2,\ 2 \xrightarrow{!} 3\,\}, 0, \{\,3\,\})$$

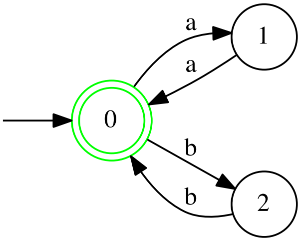


An accepting path for *baaa*!:

$$0 \overset{b}{\Longrightarrow} 1 \overset{a}{\Longrightarrow} 2 \overset{a}{\Longrightarrow} 2 \overset{a}{\Longrightarrow} 2 \overset{!}{\Longrightarrow} 3$$

Applying the Small-Step Rules, Class Exercise

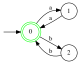
$$M = (\{\,0,1,2\,\}, \{\,0 \xrightarrow{a} 1,\ 1 \xrightarrow{a} 0,\ 0 \xrightarrow{b} 2,\ 2 \xrightarrow{b} 0\,\}, 0, \{\,0\,\})$$



What are accepting paths for *aabbaa* and *aabaa*?

Applying the Small-Step Rules, Class Exercise

$$M = (\{0,1,2\}, \{0 \xrightarrow{a} 1,\ 1 \xrightarrow{a} 0,\ 0 \xrightarrow{b} 2,\ 2 \xrightarrow{b} 0\}, 0, \{0\})$$



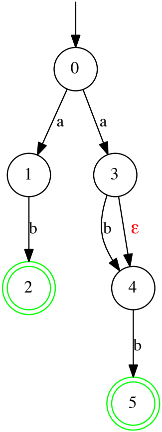
What are accepting paths for *aabbaa* and *aabaa*?

- *aabbaa* $\quad 0 \xRightarrow{a} 1 \xRightarrow{a} 0 \xRightarrow{b} 2 \xRightarrow{b} 0 \xRightarrow{a} 1 \xRightarrow{a} 0$
- *aabaa* $\quad 0 \xRightarrow{a} 1 \xRightarrow{a} 0 \xRightarrow{b} 2$ Stuck!

➤ NFAs implemented in Haskell

```
-- (trans nfa str)
--      = the set of states reachable in nfa by following str
trans :: Ord a => Nfa a -> String -> Set a
```

See http://www.cis.syr.edu/courses/cis352/code/RegExp/ImplementNfa.hs



`trans machN "a" =` $\{1,3,4\}$

- ϵ-moves are a problem
- The ϵ-closure of a *set* of states S = the *set* of states accessible from S via ϵ-moves

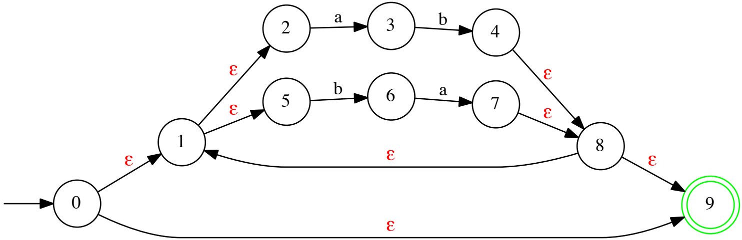
Handling ϵ-Closures

```
setlimit ::  Eq a => (Set a -> Set a) -> Set a -> Set a
setlimit f s = let next = f s
                in if s==next
                   then s
                   else setlimit f next
```

Above we assume f is *monotone*, i.e., $f(A) \supseteq A$ for all A.

```
closure ::  Ord a => Nfa a -> Set a -> Set a
closure (NFA states moves start term) s = setlimit add s
    where
      add stateset = S.union stateset (S.fromList accessible)
          where
            accessible = [ s | x <- S.toList stateset ,
                               (Emove y s) <- S.toList moves ,
                               y==x ]
```

Example: An NFA for $(ab|ba)^*$




```
*Top> closure m (singleton 2)
fromList [2]

*Top> closure m (singleton 1)
fromList [1,2,5]

*Top> closure m (singleton 0)
fromList [0,1,2,5,9]
```

Taking one step

```
onemove :: Ord a => Nfa a -> Char -> Set a -> Set a
```

```
onemove (NFA states moves start term) c x
    = S.fromList [ s | t <- S.toList x ,
                       Move z d s <- S.toList moves ,
                       z==t , c==d ]
```

$= \{ s \ : \ t \in x \ \text{ and } \ (t,c,s) \in \mathit{moves} \}$

```
onetrans :: Ord a => Nfa a -> Char -> Set a -> Set a
```

```
onetrans mach c x = closure mach (onemove mach c x)
```

Taking many steps

```
trans ::  Ord a => Nfa a -> String -> Set a
```

```
trans mach str = foldl step startset str
    where
      step set ch = onetrans mach ch set
      startset    = closure mach S.singleton (startstate mach))
```

```
foldl ::  (a -> b -> a) -> a -> [b] -> a
```

```
foldl step s (c1:c2:...:ck:[])
   = (... ((s `step` c1) `step` c2) `step` ...  `step` ck)
```

$$M(r) = \text{an NFA for accepting } L(r).$$

$$\rightarrow \textcircled{1} \xrightarrow{\epsilon} \textcircled{2}$$

Figure: $M(\epsilon)$

$$\rightarrow \textcircled{1} \xrightarrow{\mathtt{x}} \textcircled{2}$$

Figure: $M(\mathtt{x})$

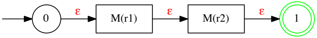


Figure: $M(r_1 r_2)$

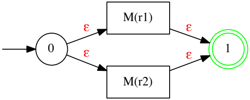


Figure: $M(r_1|r_2)$

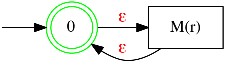
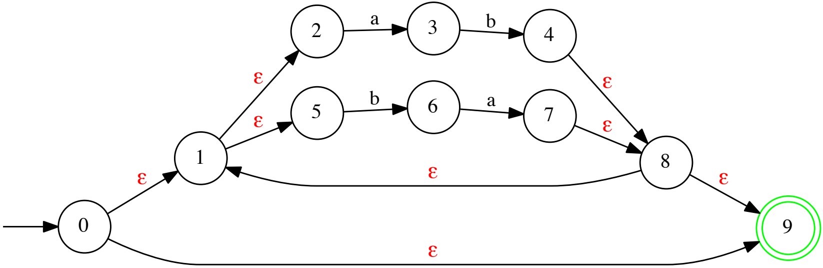


Figure: $M(r^*)$

Example: The NFA for $(ab|ba)^*$



The translation in Haskell

From: `BuildNfa.hs`

```
build :: Reg -> Nfa Int

build Epsilon        = NFA (S.fromList [0..1])
                           S.singleton(Emove 0 1)
                           0
                           S.singleton 1

build (Literal c)  = NFA (S.fromList [0..1])
                           S.singleton(Move 0 c 1)
                           0
                           S.singleton 1

build (Or r1 r2)    = m_or (build r1) (build r2)

build (Then r1 r2) = m_then (build r1) (build r2)

build (Star r)       = m_star (build r)
```

`m_or`, `m_then`, and `m_star` are a bit ugly — *because of all the state renumbering*.

Theory Break: Regular Languages

Definition 4.

The *regular languages* are the languages described by regular expressions ($= \{ L(r) : r \text{ is a reg. exp.} \}$).

Theorem 5.

The regular languages $\subseteq$ the languages accepted by NFAs.

Proof: We need to show the reg.-exp.$\rightarrow$NFA translation is correct — which is a not-too-hard structural induction.

Theorem 6.

The regular languages $\supseteq$ the languages accepted by NFAs.

Proof: There turns out to be an NFA$\rightarrow$reg.-exp. translation (which we'll skip here).

➤ Deterministic Finite Automata

Definition 7.

A *deterministic finite automata* (abbreviated DFA) is a NFA that

- contains no ϵ-moves, and
- has at most one arrow labelled with a particular symbol leaving any given state.

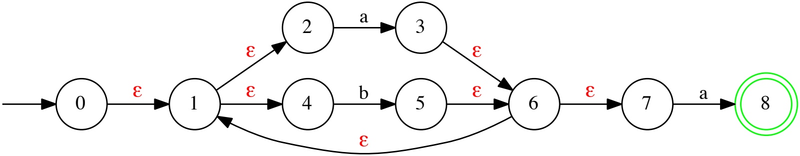
- So in a DFA there is *at most one possible move in any situation*.
- The DFAs also characterize the regular languages.

NFAs and DFAs

- They both accept exactly the regular languages.
- You can translate NFAs to equivalent DFAs, but
- you may pay a price in size blow up.

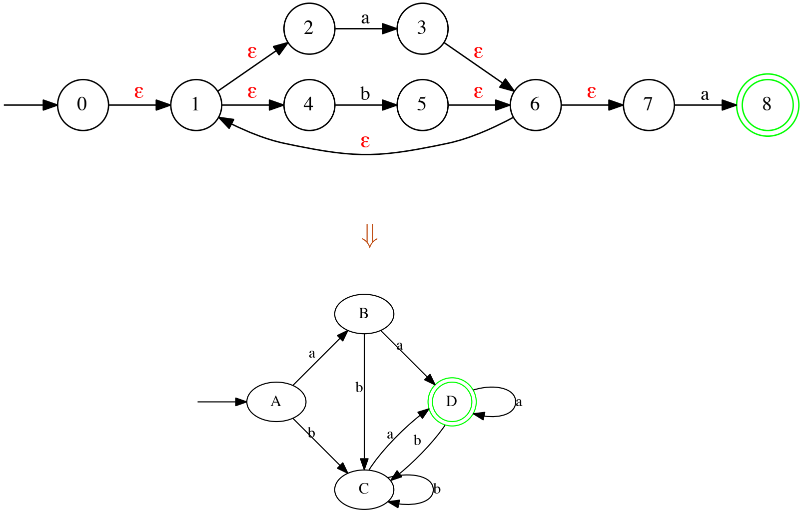
Extra Topics

Example NFA $\to$ DFA Translation, I



- $A = \epsilon\text{-closure}(\{\,0\,\}) = \{\,0, 1, 2, 4\,\}$.
- $B = \epsilon\text{-closure}(\{\,s \;:\; s' \xrightarrow{a} s,\; s' \in A\,\}) = \{\,1, 2, 3, 4, 6, 7\,\}$. $\quad (A \xrightarrow{a} B)$
- $C = \epsilon\text{-closure}(\{\,s \;:\; s' \xrightarrow{b} s,\; s' \in A\,\}) = \{\,1, 2, 4, 5, 6, 7\,\}$. $\quad (A \xrightarrow{b} C)$
- $D = \epsilon\text{-closure}(\{\,s \;:\; s' \xrightarrow{a} s,\; s' \in B\,\}) = \{\,1, 2, 4, 5, 6, 7, 8\,\}$. $\quad (B \xrightarrow{b} D)$
- $C = \epsilon\text{-closure}(\{\,s \;:\; s' \xrightarrow{b} s,\; s' \in B\,\}) = \{\,1, 2, 4, 5, 6, 7\,\}$. $\quad (B \xrightarrow{b} C)$
- Similarly, $C \xrightarrow{a} D$, $C \xrightarrow{b} C$, $D \xrightarrow{a} D$, $D \xrightarrow{b} C$.

Example NFA → DFA Translation, II



The NFA to DFA algorithm in Haskell

```
make_deterministic :: Nfa Int -> Nfa Int
make_deterministic = number . make_deter
```

```
number :: Nfa (Set Int) -> Nfa Int
number (NFA states moves start finish)
     = NFA states' moves' start' finish'
       where ...
```

```
make_deter :: Nfa Int -> Nfa (Set Int)
make_deter mach = deterministic mach (alphabet mach)
```

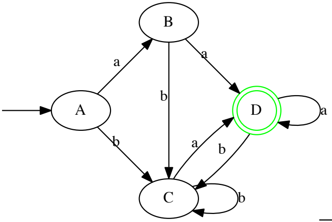
Switch to `NfaToDfa.hs`.

Minimizing DFAs, 1

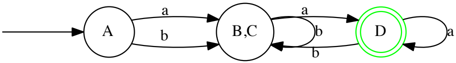
Definition 8.

Suppose s and s' are states in a DFA M.

- s and s' are distinguished by x when
  M started in s run on x accepts $\iff$ M started in s run on x rejects
- s and s' are indistinguishable when no string x distinguishes them.
  So, we can treat merge s and s' safely into a single state.



- ϵ distinguishes D and each of A, B, C
- a distinguishes A and each of B and C.
- B and C turn out to be indistinguishable.
- The result of merging B and C is:



➤ Minimizing DFAs, 2

See Tom Henzinger's notes on the Myhill-Nerode Theorem

`http://engineering.dartmouth.edu/~d25559k/ENGS122_files/Lectures_Notes/Henzinger-Nerode-7.pdf.`

(Much handier than the Pumping Lemma for regular languages)

➤ Regular Definitions, 1

- In building a compiler or interpreter, you want to specify the lexical part of the language (e.g., token) by *regular definitions* (hopped-up regular expressions). E.g.:

$$\begin{aligned} \mathit{IF} &= \texttt{if} \\ \mathit{ID} &= [\texttt{a-zA-Z}][\texttt{a-zA-Z0-9}]^* \\ \mathit{NUM} &= [-+][\texttt{0-9}]^* \\ \mathit{FLOAT} &= \textit{a nasty mess} \end{aligned}$$

- Then you translate the entire collection of these to an NFA. E.g.:

Regular Definitions, 2

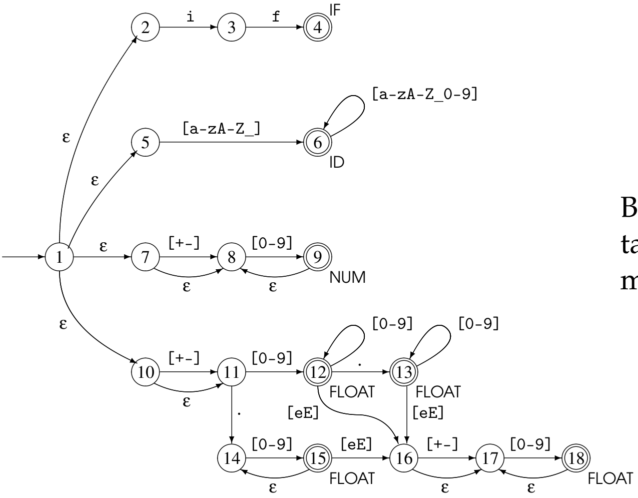


Figure 2.12: Combined NFA for several tokens

By convention, you take the longest match of a string.

Regular Definitions, 3

Then you translate the NFA to a DFA with which you scan through the input and spit out tokens with lightning speed.

See §2.9 of Mogensen for details.

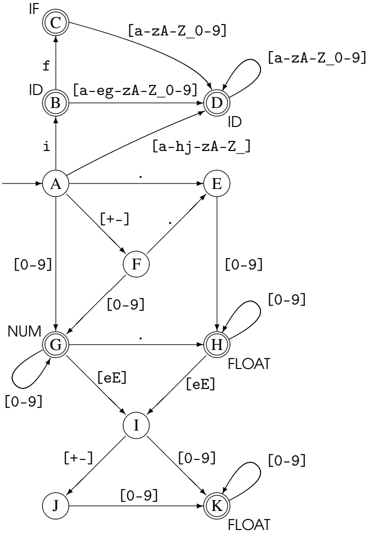


Figure 2.13: Combined DFA for several tokens

References

Also . . .

Sign up & play with the Automata Tutor: `http://www.automatatutor.com/`.