Compiler Construction

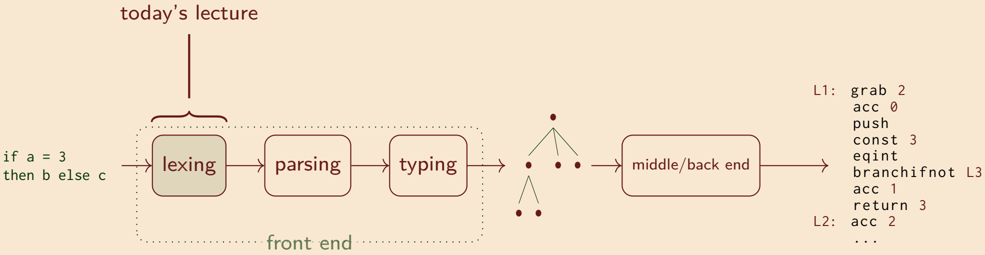
Lecture 2: Lexing

Jeremy Yallop

jeremy.yallop@cl.cam.ac.uk

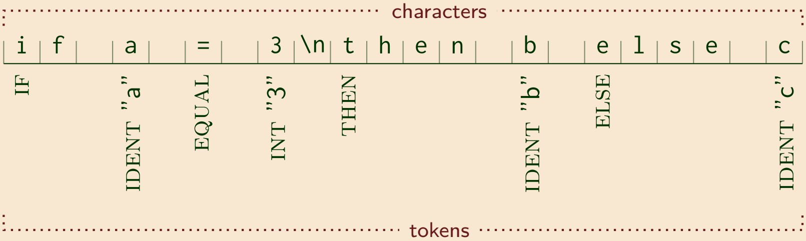
Lent 2025

What is the role of a lexer?



What is lexing?

Lexing converts a sequence of characters into a sequence of tokens.



What do lexers look like?

A **lexer** is typically specified as a sequence mapping regexes to tokens:

regular expressions		tokens
if	⇒	IF
then	⇒	THEN
else	⇒	ELSE
=	⇒	EQUAL
[a-zA-Z]+ **as** s	⇒	IDENT s
[0-9]+ **as** i	⇒	INT i
[\t\n]	⇒	*skip*

Token data type:

```
type token =
    INT of int
  | IDENT of string
  | EQUAL
  | IF
  | THEN
  | ELSE
  | ...
```

Today's Q: how can we turn this *declarative specification* into a *program*?

Regular expressions

("regexes")

Regular expression syntax

Regular expressions e over alphabet Σ are written:

$$e \to \emptyset \mid \epsilon \mid \mathsf{a} \mid e \vee e \mid ee \mid e* \qquad (\mathsf{a} \in \Sigma)$$

A regular expression e denotes a **language** (set of strings) $L(e)$. For example,

$$L((a \vee b) * abb) = \{abb, aabb, babb, aaabb, ababb, baabb, bbabb, aaaabb, \ldots\}$$

Lexing
Regexes
NFA, DFA
RE → NFA
NFA → DFA
Lexing (reprise)
∂

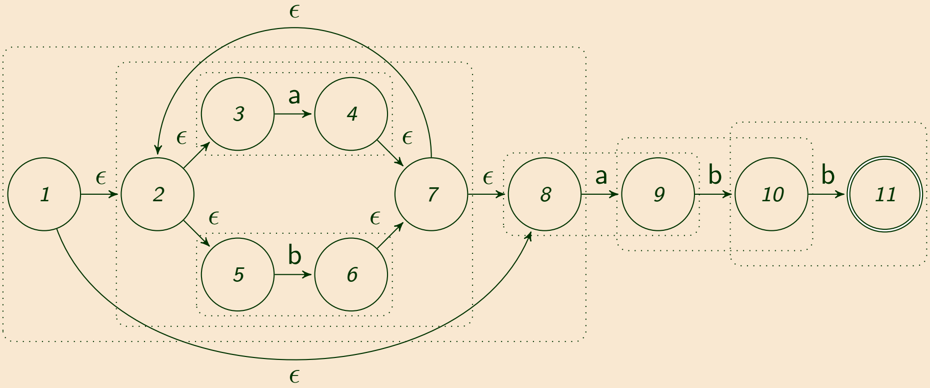
The $L(-)$ function can be defined inductively:

$$
\begin{aligned}
L(e) &\subseteq \Sigma* \\
L(\emptyset) &= \{\} \\
L(\epsilon) &= \{\epsilon\} \\
L(a) &= \{a\} \\
L(e_1 \vee e_2) &= L(e_1) \cup L(e_2) \\
L(e_1 e_2) &= \{w_1 w_2 \mid w_1 \in L(e_1), w_2 \in L(e_2)\} \\
L(e^0) &= \{\epsilon\} \\
L(e^{n+1}) &= L(ee^n) \\
L(e*) &= \cup_{n \geq 0} L(e^n)
\end{aligned}
$$

The **regular language problem**: is $w \in L(e)$? This is **insufficient for lexing**.

Finite-state automata

A nondeterministic finite-state automaton for recognising $L((a \vee b) * abb)$:



Review of Finite Automata (FA)

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

- states Q
- start state $q_0 \in Q$
- alphabet Σ
- final states $F \subseteq Q$

For NFAs: (**nondeterministic**)
$\forall q \in Q$
  $\forall a \in (\Sigma \cup \{\epsilon\})$
    $\delta(q, a) \subseteq Q$

For DFAs: (**deterministic**)
$\forall q \in Q$
  $\forall a \in \Sigma$
    $\delta(q, a) \in Q$

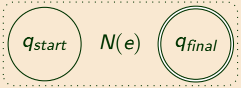
Transition notation

DFA		NFA
$q \xrightarrow{\epsilon} q$	Null transition on empty string	$q \xrightarrow{\epsilon} q$
	Including ϵ transitions	$q_1 \xrightarrow{w} q_3$ if $\delta(q_1, \epsilon) \ni q_2$ and $q_2 \xrightarrow{w} q_3$
$q_1 \xrightarrow{aw} q_3$ if $\delta(q_1, a) = q_2$ and $q_2 \xrightarrow{w} q_3$	Transition on non-empty string	$q_1 \xrightarrow{aw} q_3$ if $\delta(q_1, a) \ni q_2$ and $q_2 \xrightarrow{w} q_3$
$L(M) = \{w \mid \exists q \in F, q_0 \xrightarrow{w} q\}$	Language of an automaton	$L(M) = \{w \mid \exists q \in F, q_0 \xrightarrow{w} q\}$

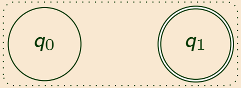
Regular expressions $\longrightarrow$ NFAs

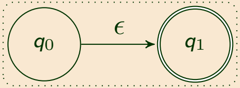
Review of RE $\longrightarrow$ NFA

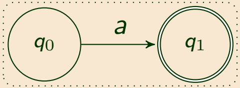
$N(-)$ takes a regex e to an NFA $N(e)$ accepting $L(e)$ with a single final state.

$N(e) =$ 

$N(-)$ is defined by induction on e.

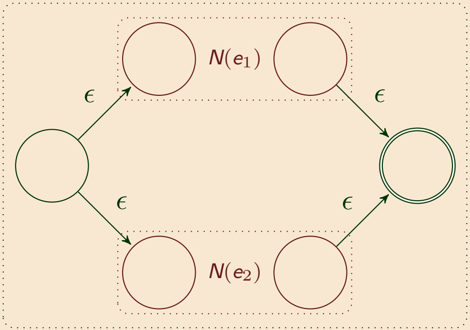
$N(\emptyset) =$ 

$N(\epsilon) =$ 

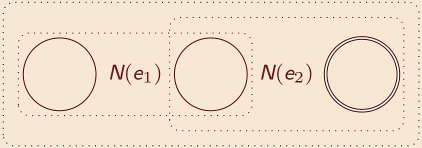
$N(a) =$ 

Review of RE ⟶ NFA

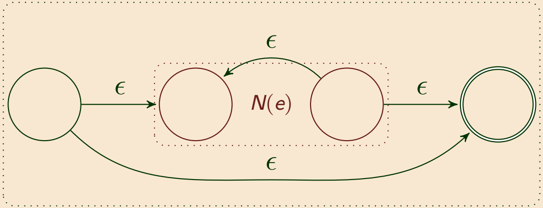
$N(e_1 \vee e_2) \quad =$



$N(e_1 e_2) \quad =$



$N(e*) \quad =$



Note: *an* ***alternative*** *to this simple construction is* ***Glushkov's algorithm*** *(1961), which produces an equivalent automaton without the* ϵ *transitions.*

NFAs $\longrightarrow$ DFAs

Review of NFA ⟶ DFA

The **powerset construction** takes a NFA

$$M = \langle Q, \Sigma, \delta, q_0, F \rangle$$

and constructs a DFA

$$M' = \langle Q', \Sigma', \delta', q'_0, F' \rangle$$

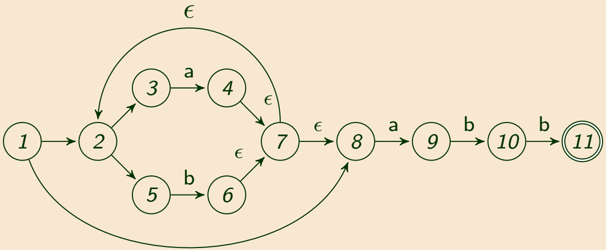
where the components of M' are calculated as follows:

$$\begin{aligned}
Q' &= \{S \mid S \subseteq Q\} \\
\delta'(S, a) &= \epsilon\text{-closure}(\{q' \in \delta(q, a) \mid q \in S\}) \\
q'_0 &= \epsilon\text{-closure}\{q_0\} \\
F' &= \{S \subseteq Q \mid S \cap F \neq \emptyset\}
\end{aligned}$$

and the ϵ-*closure* is:

$$\epsilon\text{-closure}(S) = \{q' \in Q \mid \exists q \in S, q \xrightarrow{\epsilon} q'\}$$

How do we compute ϵ-closure(S)?



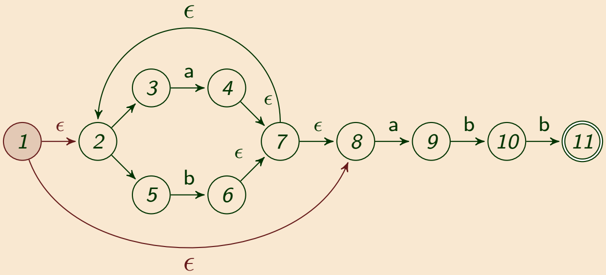
ϵ-closure

```
push elements of S onto stack
result := S
while stack not empty
   pop q off stack
   for each u ∈ δ(q, ϵ)
      if u ∉ result
      then result := {u}∪ result
         push u on stack
return result
```

stack	
result	

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



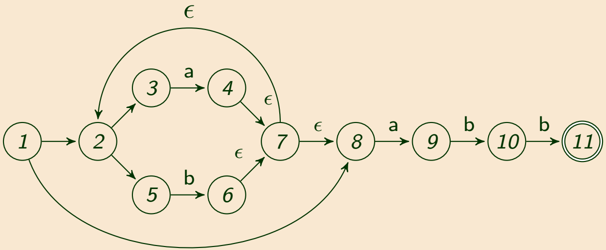
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	1
result	1

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



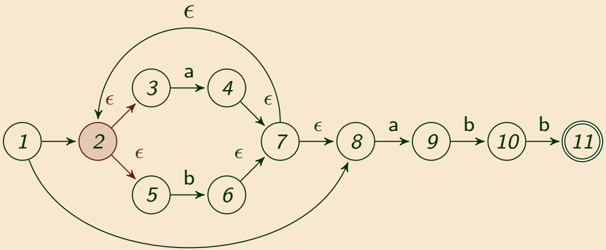
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	
result	1 2 8

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



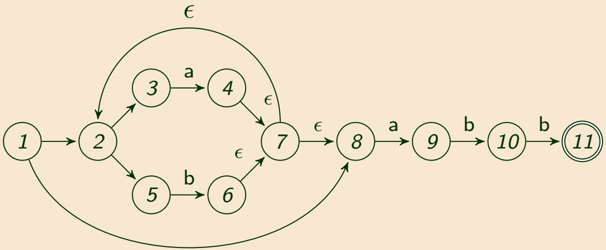
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	2 8
result	1 2 8

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



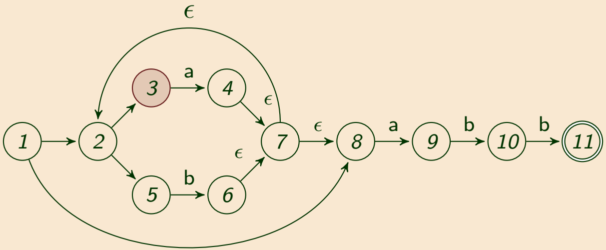
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	8
result	1 2 8 3 5

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



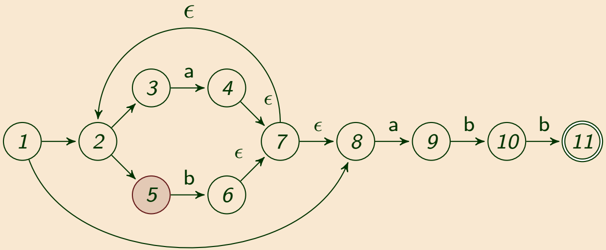
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	3 5 8
result	1 2 8 3 5

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



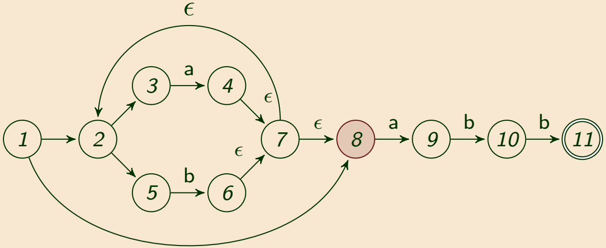
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	5 8
result	1 2 8 3 5

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



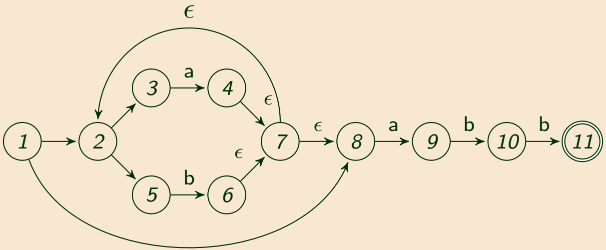
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	8
result	1 2 8 3 5

(NB: just an instance of **transitive closure**)

How do we compute ϵ-closure(S)?



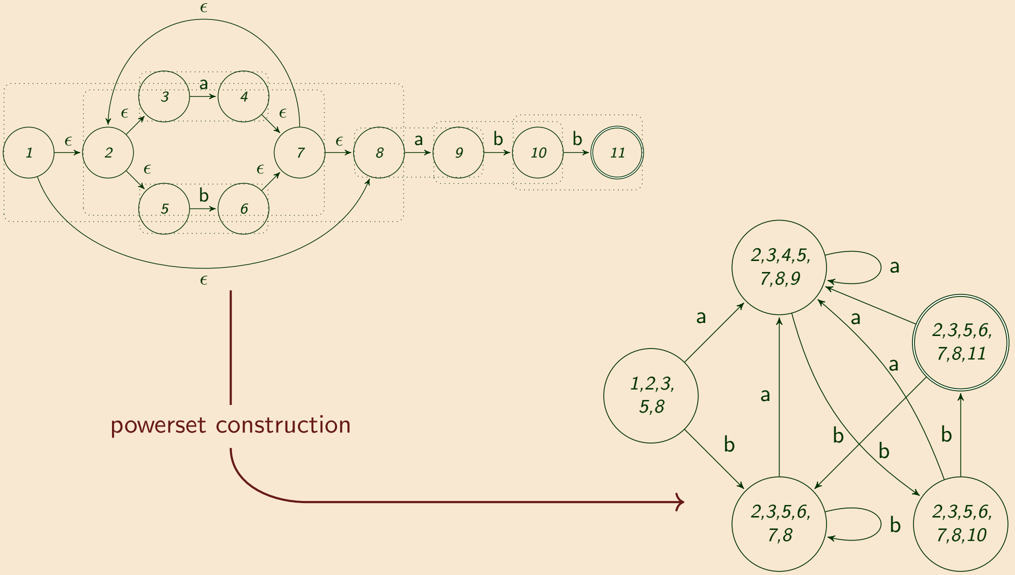
ϵ-closure

push elements of S onto stack
result := S
while stack not empty
  pop q off stack
  for each $u \in \delta(q, \epsilon)$
    if $u \notin$ result
    then result := $\{u\} \cup$ result
      push u on stack
return result

stack	
result	1 2 8 3 5

(NB: just an instance of **transitive closure**)

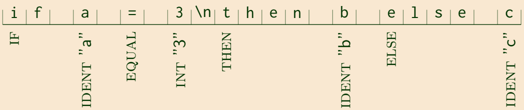
DFA(N($(a \vee b) * abb$))



The lexing problem

The lexing problem

The regular language problem (i.e. "is $w \in L(e)$?") is insufficient for lexing.
We need to tokenize a string using a lexer specification



if	⇒	IF
...		
[a-zA-Z]+ **as** s	⇒	IDENT s
[0-9]+ **as** i	⇒	INT i
[\t\n]	⇒	*skip*

taking into account that

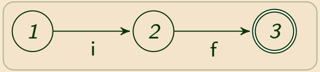
We should skip whitespace
(because whitespace is irrelevant to the parser)

We should find the longest match accepted by the lexer
(treat `ifif` as a variable, not two keywords)

We should pick the first rule that matches the longest matched substring
(treat `if` as a keyword because the IF rule comes before the IDENT rule)

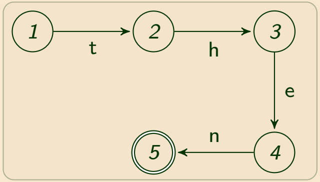
Define tokens with regexes (automata)

`if`



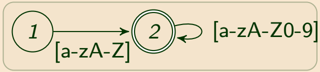
⇒ IF

`then`



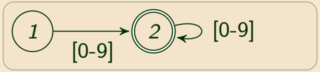
⇒ THEN

`[a-zA-Z][a-zA-Z0-9]*`



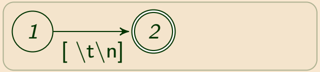
⇒ IDENT s

`[0-9][0-9]*`



⇒ INT n

`[ \t\n]`



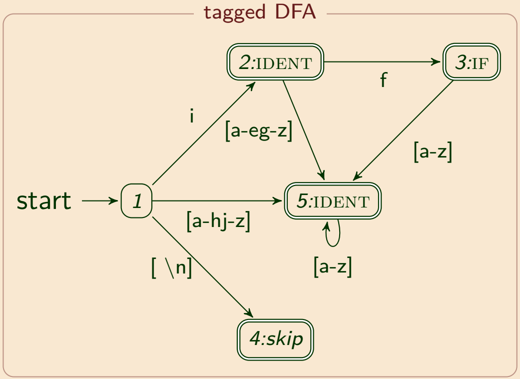
⇒ *skip*
(not really a token)

Constructing a Lexer

Start from ordered lexer rules $e_1 \Rightarrow t_1, e_2 \Rightarrow t_2, \ldots, e_k \Rightarrow t_k$.
Construct *tagged NFA* for $e_1 \vee e_2 \vee \ldots \vee e_k$.
Convert to *tagged DFA*: each accepting state is tagged for highest priority e_i.

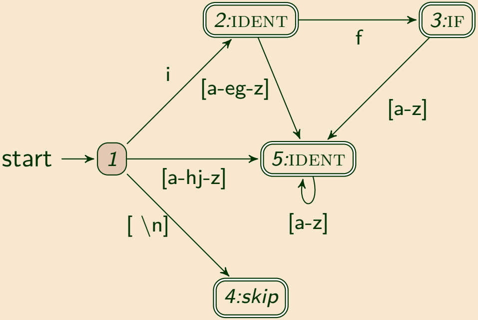
lexer rules

if	⇒	IF
...		
[a-zA-Z]+ **as** s	⇒	IDENT s
[0-9]+ **as** i	⇒	INT i
[\n]	⇒	*skip*



State 3 could be either an IDENT or the keyword IF.
Priority eliminates the ambiguity, associating state 3 with the keyword.

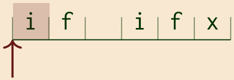
What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position

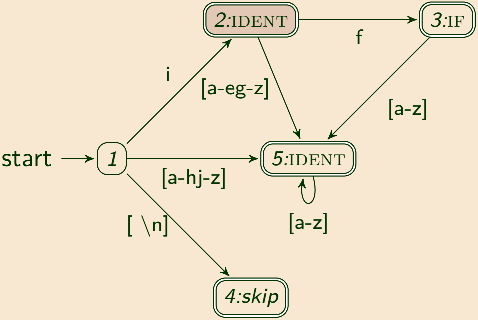


tokens:

Note: the machine is deterministic, but **the algorithm can backtrack**.

What about longest match?

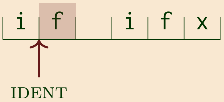
Lexing
Regexes
NFA, DFA
RE → NFA
NFA → DFA
Lexing (reprise)
∂



lexing algorithm

Start in initial state, and repeatedly:

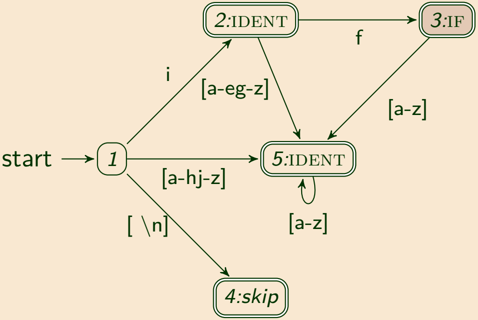
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens:

Note: the machine is deterministic, but **the algorithm can backtrack**.

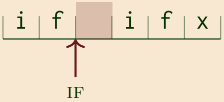
What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

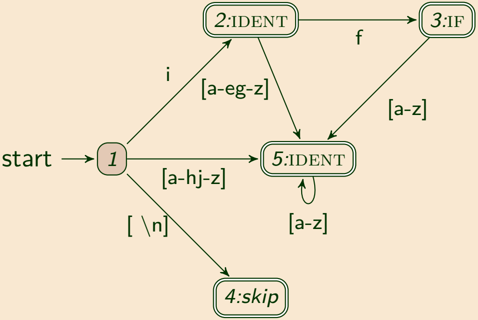
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens:

Note: the machine is deterministic, but **the algorithm can backtrack**.

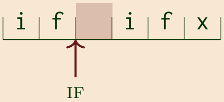
What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

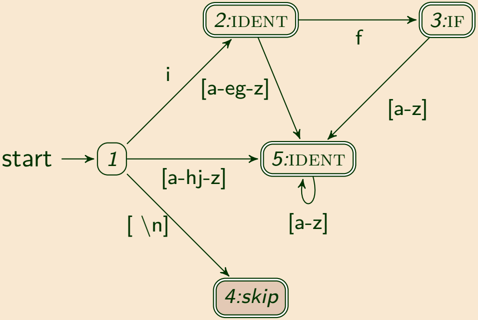
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens: IF

Note: the machine is deterministic, but **the algorithm can backtrack**.

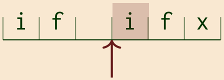
What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

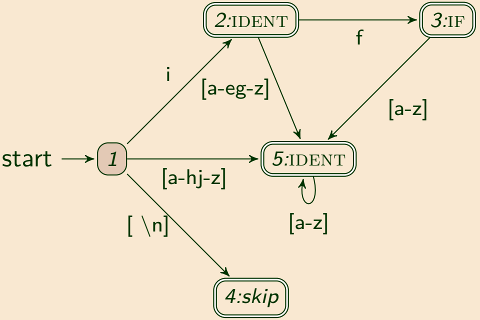
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens: IF

Note: the machine is deterministic, but **the algorithm can backtrack**.

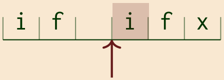
What about longest match?



lexing algorithm

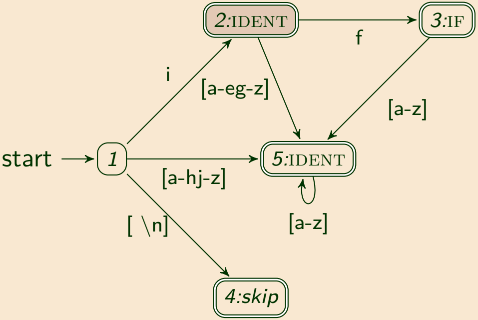
Start in initial state, and repeatedly:

1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens: IF

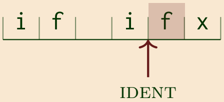
Note: the machine is deterministic, but **the algorithm can backtrack**.



lexing algorithm

Start in initial state, and repeatedly:

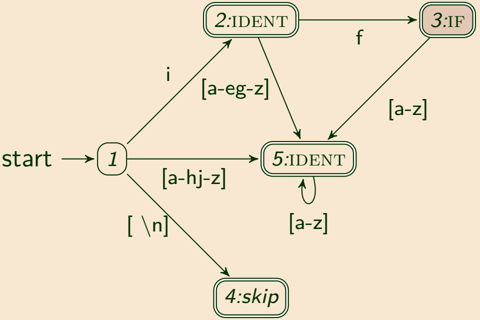
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens: IF

Note: the machine is deterministic, but **the algorithm can backtrack**.

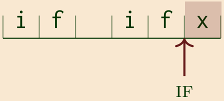
What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

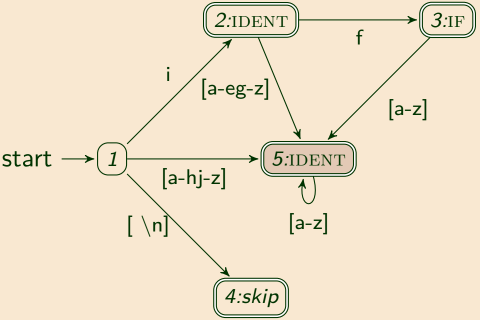
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens: IF

Note: the machine is deterministic, but **the algorithm can backtrack**.

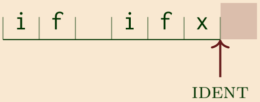
What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

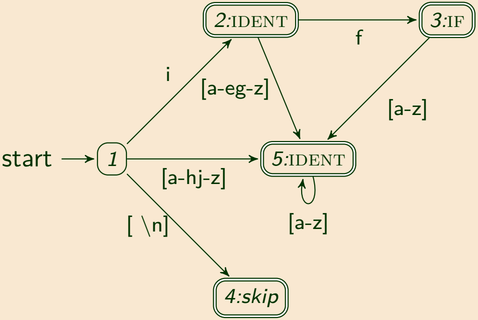
1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position



tokens: IF

Note: the machine is deterministic, but **the algorithm can backtrack**.

What about longest match?



lexing algorithm

Start in initial state, and repeatedly:

1. Read input until failure (no transition)
   Emit tag for last accepting state
2. Reset state to start state
   Reset position to last accepting position

| i | f | | i | f | x |

tokens: IF IDENT `ifx`

Note: the machine is deterministic, but **the algorithm can backtrack**.

Lexing with derivatives

Matching with derivatives

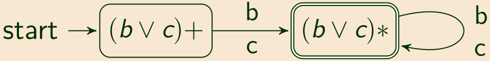
Brzozowski (1964)'s formulation of regex matching, based on *derivatives*.

Derivative of regex r **w.r.t. character** c is
another regex $\partial_c\, r$ that matches s iff r matches cs.

E.g.: consider $(b \vee c)+$. After matching c, can accept either ϵ or more b/c, so:

$$\partial_c\,(b \vee c)+ \quad = \quad \epsilon \vee (b \vee c)+ \quad = \quad (b \vee c)*$$

Construct DFA for r, taking regexes r as states, adding transition $r_i \xrightarrow{c} r_j$ whenever $\partial_c\, r_i = r_j$. For example, for $(b \vee c)+$:



NB: $\partial_c\,(b \vee c)* = (b \vee c)*$. (Can you see why?) Also: ϵ-matching states are accepting.

Defining ∂_c

∂_c is defined inductively over regexes.

Can you see the similarities with derivatives of numerical functions?
(Hint: read $r_1 r_2$ as $r_1 \times r_2$ and $r_1 \vee r_2$ as $r_1 + r_2$.)

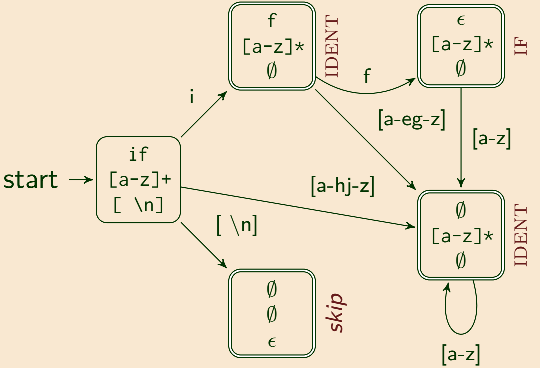
$$
\begin{aligned}
\partial_c\, \emptyset &= \emptyset \\
\partial_c\, \epsilon &= \emptyset \\
\partial_c\, b &= \emptyset \\
\partial_c\, c &= \epsilon \\
\partial_c\, (rs) &= (\partial_c\, r)s \mid \nu(r)(\partial_c\, s) \\
\partial_c\, (r \vee s) &= \partial_c\, r \vee \partial_c\, s \\
\partial_c\, r* &= (\partial_c\, r)r*
\end{aligned}
$$

$$
\begin{aligned}
\nu(r) &= \epsilon \text{ if } \epsilon \in L(r) \\
&= \emptyset \text{ if } \epsilon \notin L(r)
\end{aligned}
$$

More information: Regular-expression derivatives re-examined (Owens et al, 2009).

Lexing
Regexes
NFA, DFA
RE → NFA
NFA → DFA
Lexing (reprise)
∂

Lexers match input string against multiple regexes in parallel.
Automaton for matching a token; states are vectors of regexes, one per lexer rule.
∂_c acts pointwise on the regex vector.



Next time: context-free grammars