Lexical Analysis

Compiler Design

CSE 504

Last modified: Fri Feb 12 2016 at 12:27:04 EST
Version: 1.6 16:58:46 2016/01/29
Compiled at 12:48 on 2016/02/12

Structure of a Language

Grammars: Notation to succinctly represent the structure of a language.
Example:

$$
\begin{array}{lcl}
\mathit{Stmt} & \longrightarrow & \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Stmt}\ \texttt{else}\ \mathit{Stmt} \\
\mathit{Stmt} & \longrightarrow & \texttt{while}\ \mathit{Expr}\ \texttt{do}\ \mathit{Stmt} \\
\mathit{Stmt} & \longrightarrow & \texttt{do}\ \mathit{Stmt}\ \texttt{until}\ \mathit{Expr} \\
\vdots & & \\
\mathit{Expr} & \longrightarrow & \mathit{Expr}\ \texttt{+}\ \mathit{Expr} \\
\vdots & &
\end{array}
$$

Grammars

$$\textit{Stmt} \longrightarrow \texttt{if}\ \textit{Expr}\ \texttt{then}\ \textit{Stmt}\ \texttt{else}\ \textit{Stmt}$$

- Terminal symbols: `if`, `then`, `else`
  - Terminal symbols represent group of characters in input language: *Tokens*.
  - Analogous to *words*.
- Nonterminal symbols: *Stmt*, *Expr*
  - Nonterminal symbols represent a sequence of terminal symbols.
  - Analogous to *sentences*.

Phases of Syntax Analysis

1. Identify the words: **Lexical Analysis**.
   Converts a stream of characters (input program) into a stream of tokens.
   Also called *Scanning* or *Tokenizing*.
2. Identify the sentences: **Parsing**.
   Derive the structure of sentences: construct *parse trees* from a stream of tokens.

Lexical Analysis

Convert a stream of characters into a stream of *tokens*.

- Simplicity: Conventions about "words" are often different from conventions about "sentences".
- Efficiency: Word identification problem has a much more efficient solution than sentence identification problem.
- Portability: Character set, special characters, device features.

Terminology

- Token: Name given to a family of words.
  e.g., `integer_constant`
- Lexeme: Actual sequence of characters representing a word.
  e.g., 32894
- Pattern: Notation used to identify the set of lexemes represented by a token.
  e.g., $[0-9]+$

Terminology

A few more examples:

Token	Sample Lexemes	Pattern
`while`	`while`	`while`
`integer_constant`	32894, -1093, 0	[0-9]+
`identifier`	`buffer_size`	[a-zA-Z]+

Patterns

How do we *compactly* represent the set of all lexemes corresponding to a token?

For instance:

> *The token* `integer_constant` *represents the set of all integers: that is, all sequences of digits (0–9), preceded by an optional sign (+ or −).*

Obviously, we cannot simply enumerate all lexemes.

Use **Regular Expressions**.

Regular Expressions

Notation to represent (potentially) infinite sets of strings over alphabet Σ.

- a: stands for the set $\{\mathtt{a}\}$ that contains a single string $\mathtt{a}$.
- $a \mid b$: stands for the set $\{\mathtt{a}, \mathtt{b}\}$ that contains two strings $\mathtt{a}$ and $\mathtt{b}$.
  - Analogous to *Union*.
- ab: stands for the set $\{\mathtt{ab}\}$ that contains a single string $\mathtt{ab}$.
  - Analogous to *Product*.
  - $(a|b)(a|b)$: stands for the set $\{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}$.
- a^*: stands for the set $\{\epsilon, \mathtt{a}, \mathtt{aa}, \mathtt{aaa}, \ldots\}$ that contains all strings of zero or more $\mathtt{a}$'s.
  - Analogous to *closure* of the product operation.

ϵ stands for the *empty string*.

Regular Expressions

Examples of Regular Expressions over $\{\mathtt{a}, \mathtt{b}\}$:

- $(a|b)^*$: Set of strings with zero or more a's and zero or more b's:
  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$
- $(a^* b^*)$: Set of strings with zero or more a's and zero or more b's such that all a's occur before any b:
  $\{\epsilon, a, b, aa, ab, bb, aaa, aab, abb, \ldots\}$
- $(a^* b^*)^*$: Set of strings with zero or more a's and zero or more b's:
  $\{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$

Language of Regular Expressions

Let R be the set of all regular expressions over Σ. Then,

- Empty String: $\epsilon \in R$
- Unit Strings: $\alpha \in \Sigma \Rightarrow \alpha \in R$
- Concatenation: $r_1, r_2 \in R \Rightarrow r_1 r_2 \in R$
- Alternative: $r_1, r_2 \in R \Rightarrow (r_1 \mid r_2) \in R$
- Kleene Closure: $r \in R \Rightarrow r^* \in R$

Regular Expressions

Example: $(a \mid b)^*$

$$
\begin{aligned}
L_0 &= \{\epsilon\} \\
L_1 &= L_0 \cdot \{\mathtt{a}, \mathtt{b}\} \\
&= \{\epsilon\} \cdot \{\mathtt{a}, \mathtt{b}\} \\
&= \{\mathtt{a}, \mathtt{b}\} \\
L_2 &= L_1 \cdot \{\mathtt{a}, \mathtt{b}\} \\
&= \{\mathtt{a}, \mathtt{b}\} \cdot \{\mathtt{a}, \mathtt{b}\} \\
&= \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\} \\
L_3 &= L_2 \cdot \{\mathtt{a}, \mathtt{b}\} \\
&\vdots
\end{aligned}
$$

$$
L = \bigcup_{i=0}^{\infty} L_i \qquad = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \ldots\}
$$

Semantics of Regular Expressions

Semantic Function $\mathcal{L}$: Maps regular expressions to sets of strings.

$$
\begin{array}{rcll}
\mathcal{L}(\epsilon) & = & \{\epsilon\} & \\
\mathcal{L}(\alpha) & = & \{\alpha\} & (\alpha \in \Sigma) \\
\mathcal{L}(r_1 \mid r_2) & = & \mathcal{L}(r_1) \cup \mathcal{L}(r_2) & \\
\mathcal{L}(r_1\ r_2) & = & \mathcal{L}(r_1) \cdot \mathcal{L}(r_2) & \\
\mathcal{L}(r^*) & = & \{\epsilon\} \cup (\mathcal{L}(r) \cdot \mathcal{L}(r^*)) &
\end{array}
$$

Computing the Semantics

$$
\begin{aligned}
\mathcal{L}(a) &= \{\mathtt{a}\} \\
\mathcal{L}(a \mid b) &= \mathcal{L}(a) \cup \mathcal{L}(b) \\
&= \{\mathtt{a}\} \cup \{\mathtt{b}\} \\
&= \{\mathtt{a}, \mathtt{b}\} \\
\mathcal{L}(ab) &= \mathcal{L}(a) \cdot \mathcal{L}(b) \\
&= \{\mathtt{a}\} \cdot \{\mathtt{b}\} \\
&= \{\mathtt{ab}\} \\
\mathcal{L}((a \mid b)(a \mid b)) &= \mathcal{L}(a \mid b) \cdot \mathcal{L}(a \mid b) \\
&= \{\mathtt{a}, \mathtt{b}\} \cdot \{\mathtt{a}, \mathtt{b}\} \\
&= \{\mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\}
\end{aligned}
$$

Computing the Semantics of Closure

Example: $\mathcal{L}((a \mid b)^*)$
$= \{\epsilon\} \cup (\mathcal{L}(a \mid b) \cdot \mathcal{L}((a \mid b)^*))$

$$
\begin{array}{lcll}
L_0 & = & \{\epsilon\} & \textit{Base case} \\
L_1 & = & \{\epsilon\} \cup (\{\mathtt{a},\mathtt{b}\} \cdot L_0) & \\
 & = & \{\epsilon\} \cup (\{\mathtt{a},\mathtt{b}\} \cdot \{\epsilon\}) & \\
 & = & \{\epsilon, \mathtt{a}, \mathtt{b}\} & \\
L_2 & = & \{\epsilon\} \cup (\{\mathtt{a},\mathtt{b}\} \cdot L_1) & \\
 & = & \{\epsilon\} \cup (\{\mathtt{a},\mathtt{b}\} \cdot \{\epsilon, \mathtt{a}, \mathtt{b}\}) & \\
 & = & \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}\} & \\
\vdots & & &
\end{array}
$$

$$\mathcal{L}((a \mid b)^*) = L_\infty = \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \ldots\}$$

Another Example

$\mathcal{L}((\mathbf{a}^*\mathbf{b}^*)^*)$:

$$
\begin{aligned}
\mathcal{L}(a^*) &= \{\epsilon, \mathrm{a}, \mathrm{aa}, \ldots\} \\
\mathcal{L}(b^*) &= \{\epsilon, \mathrm{b}, \mathrm{bb}, \ldots\} \\
\mathcal{L}(a^* b^*) &= \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{bb}, \\
&\qquad \mathrm{aaa}, \mathrm{aab}, \mathrm{abb}, \mathrm{bbb}, \ldots\} \\
\mathcal{L}((a^* b^*)^*) &= \{\epsilon\} \\
&\quad \cup\{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{bb}, \\
&\qquad \mathrm{aaa}, \mathrm{aab}, \mathrm{abb}, \mathrm{bbb}, \ldots\} \\
&\quad \cup\{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}, \\
&\qquad \mathrm{aaa}, \mathrm{aab}, \mathrm{aba}, \mathrm{abb}, \mathrm{baa}, \mathrm{bab}, \mathrm{bba}, \mathrm{bbb}, \ldots\} \\
&\quad \vdots \\
&= \{\epsilon, \mathrm{a}, \mathrm{b}, \mathrm{aa}, \mathrm{ab}, \mathrm{ba}, \mathrm{bb}, \ldots\}
\end{aligned}
$$

Regular Definitions

Assign "names" to regular expressions.
For example,

$$\begin{array}{rcl} \texttt{digit} & \longrightarrow & 0 \mid 1 \mid \cdots \mid 9 \\ \texttt{natural} & \longrightarrow & \texttt{digit}\ \texttt{digit}^* \end{array}$$

SHORTHANDS:

- a^+: Set of strings with one or more occurrences of a.
- $a^?$: Set of strings with zero or one occurrences of a.

Example:

$$\texttt{integer} \longrightarrow (+|-)^?\texttt{digit}^+$$

Regular Definitions: Examples

$$
\begin{array}{rcl}
\texttt{float} & \longrightarrow & \texttt{integer} \; . \; \texttt{fraction} \\
\texttt{integer} & \longrightarrow & (+|-)^? \; \texttt{no\_leading\_zero} \\
\texttt{no\_leading\_zero} & \longrightarrow & (\texttt{nonzero\_digit} \; \texttt{digit}^*) \mid 0 \\
\texttt{fraction} & \longrightarrow & \texttt{no\_trailing\_zero} \; \texttt{exponent}^? \\
\texttt{no\_trailing\_zero} & \longrightarrow & (\texttt{digit}^* \; \texttt{nonzero\_digit}) \mid 0 \\
\texttt{exponent} & \longrightarrow & (\texttt{E} \mid \texttt{e}) \; \texttt{integer} \\
\texttt{digit} & \longrightarrow & 0 \mid 1 \mid \cdots \mid 9 \\
\texttt{nonzero\_digit} & \longrightarrow & 1 \mid 2 \mid \cdots \mid 9
\end{array}
$$

Regular Definitions and Lexical Analysis

Regular Expressions and Definitions *specify* sets of strings over an input alphabet.

- They can hence be used to specify the set of *lexemes* associated with a *token*.
- That is, regular expressions and definitions can be used as the *pattern* language

How do we decide whether an input string belongs to the set of strings specified by a regular expression?

Using Regular Definitions for Lexical Analysis

Q: Is ababbaabbb in $\mathcal{L}(((a^*b^*)^*)$?
A: Hm. Well. Let's see.

$$
\begin{aligned}
\mathcal{L}((a^*b^*)^*) &= \{\epsilon\} \\
&\quad \cup \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{bb}, \\
&\qquad \mathtt{aaa}, \mathtt{aab}, \mathtt{abb}, \mathtt{bbb}, \ldots\} \\
&\quad \cup \{\epsilon, \mathtt{a}, \mathtt{b}, \mathtt{aa}, \mathtt{ab}, \mathtt{ba}, \mathtt{bb}, \\
&\qquad \mathtt{aaa}, \mathtt{aab}, \mathtt{aba}, \mathtt{abb}, \mathtt{baa}, \mathtt{bab}, \mathtt{bba}, \mathtt{bbb}, \ldots\} \\
&\quad \vdots \\
&= ???
\end{aligned}
$$

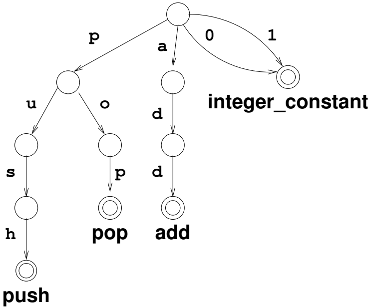
Recognizers

Construct *automata* that recognize strings belonging to a language.

- Finite State Automata $\Rightarrow$ Regular Languages
  - Finite State $\rightarrow$ cannot maintain arbitrary counts.
- Push Down Automata $\Rightarrow$ Context-free Languages
  - Stack is used to maintain counter, but only one counter can go arbitrarily high.

Recognizing Finite Sets of Strings

- Identifying words from a small, finite, fixed vocabulary is straightforward.
- For instance, consider a stack machine with `push`, `pop`, and `add` operations with two constants: `0` and `1`.
- We can use the *automaton*:



Finite State Automata

Represented by a labeled directed graph.

- A finite set of *states* (vertices).
- *Transitions* between states (edges).
- *Labels* on transitions are drawn from $\Sigma \cup \{\epsilon\}$.
- One distinguished *start* state.
- One or more distinguished *final* states.

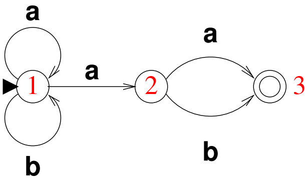
Finite State Automata: An Example

Consider the Regular Expression **(a | b)*a(a | b)**.

$\mathcal{L}((a \mid b)^* a(a \mid b)) = \{\texttt{aa}, \texttt{ab}, \texttt{aaa}, \texttt{aab}, \texttt{baa}, \texttt{bab}, \texttt{aaaa}, \texttt{aaab}, \texttt{abaa}, \texttt{abab}, \texttt{baaa}, \ldots\}$.

Finite State Automata: An Example

Consider the Regular Expression **(a | b)*a(a | b)**.
$\mathcal{L}((a \mid b)^* a(a \mid b)) = \{\mathtt{aa}, \mathtt{ab}, \mathtt{aaa}, \mathtt{aab}, \mathtt{baa}, \mathtt{bab}, \mathtt{aaaa}, \mathtt{aaab}, \mathtt{abaa}, \mathtt{abab}, \mathtt{baaa}, \ldots\}$.
The following automaton determines whether an input string belongs to $\mathcal{L}((a \mid b)^* a(a \mid b)$:



Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string x

. . . if beginning from the start state

Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string x

... if beginning from the start state

... we can trace some path through the automaton

Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string x

... if beginning from the start state

... we can trace some path through the automaton

... such that the sequence of edge labels spells x

Acceptance Criterion

A finite state automaton (NFA or DFA) *accepts* an input string x

... if beginning from the start state

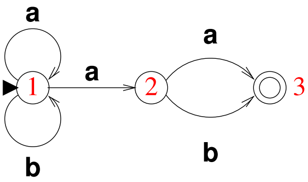
... we can trace some path through the automaton

... such that the sequence of edge labels spells x

... and end in a final state.

Recognition with an NFA

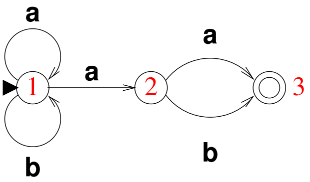
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	no

Recognition with an NFA

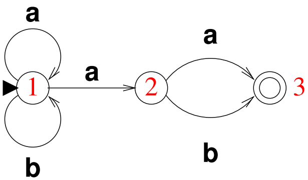
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	no
Path 2:	1	1	1	2	3	yes

Recognition with an NFA

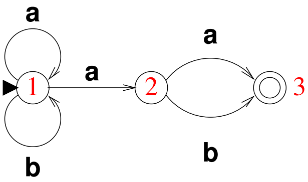
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	no
Path 2:	1	1	1	2	3	yes
Path 3:	1	2	3	$\perp$	$\perp$	no

Recognition with an NFA

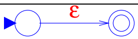
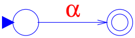
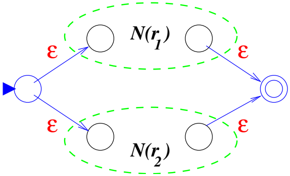
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	no
Path 2:	1	1	1	2	3	yes
Path 3:	1	2	3	$\perp$	$\perp$	no
						YES

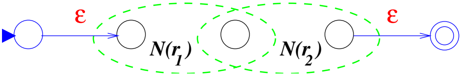
Regular Expressions to NFA

Thompson's Construction: For every regular expression r, derive an NFA $N(r)$ with unique start and final states.

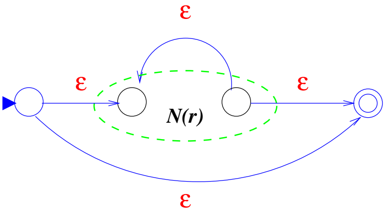
ϵ	
$\alpha \in \Sigma$	
$(r_1 \mid r_2)$	

Regular Expressions to NFA (contd.)

$r_1 r_2$

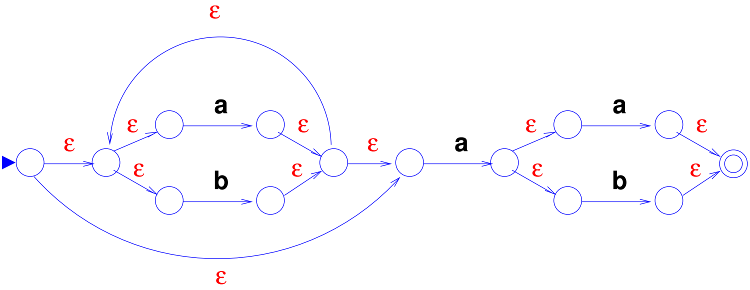


r^*



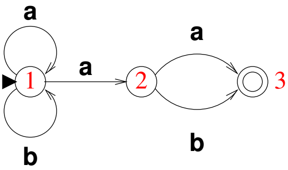
Example

(a | b)*a(a | b):



Recognition with an NFA

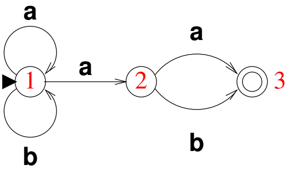
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	

Recognition with an NFA

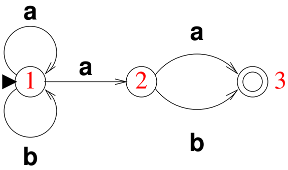
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept

Recognition with an NFA

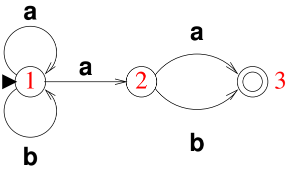
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	

Recognition with an NFA

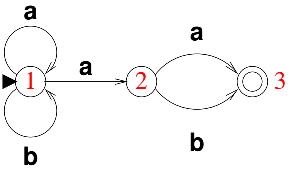
Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,3\}$	$\{1,2\}$	$\{1,3\}$	Accept

Recognition with an NFA (contd.)

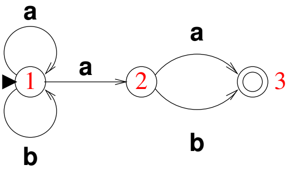
Is aaab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	a	b	Accept?
Path 1:	1	1	1	1	1	

Recognition with an NFA (contd.)

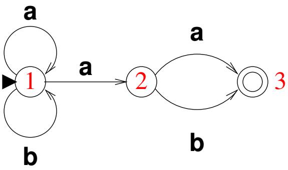
Is aaab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept

Recognition with an NFA (contd.)

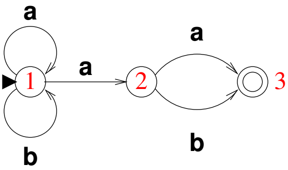
Is aaab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	1	2	3	$\perp$	

Recognition with an NFA (contd.)

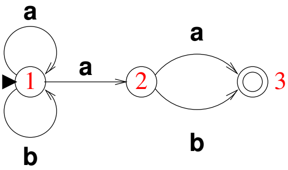
Is aaab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	1	2	3	$\perp$	
Path 4:	1	2	3	$\perp$	$\perp$	

Recognition with an NFA (contd.)

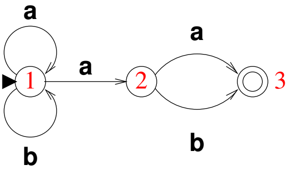
Is aaab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	a	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	1	2	3	Accept
Path 3:	1	1	2	3	$\perp$	
Path 4:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,2,3\}$	$\{1,3\}$	Accept

Recognition with an NFA (contd.)

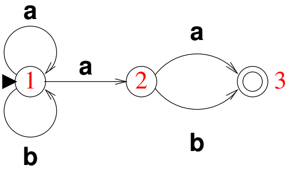
Is aabb $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	b	b	Accept?
Path 1:	1	1	1	1	1	

Recognition with an NFA (contd.)

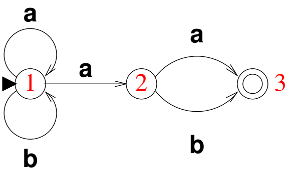
Is aabb $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	b	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	2	3	$\perp$	

Recognition with an NFA (contd.)

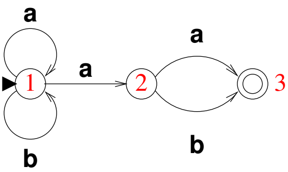
Is aabb $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	a	b	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	2	3	$\perp$	
Path 3:	1	2	3	$\perp$	$\perp$	

Recognition with an NFA (contd.)

Is aabb $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?

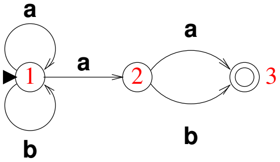


Input:		a	a	b	b	Accept?
Path 1:	1	1	1	1	1	
Path 2:	1	1	2	3	$\perp$	
Path 3:	1	2	3	$\perp$	$\perp$	
All Paths	$\{1\}$	$\{1,2\}$	$\{1,2,3\}$	$\{1,3\}$	$\{1\}$	REJECT

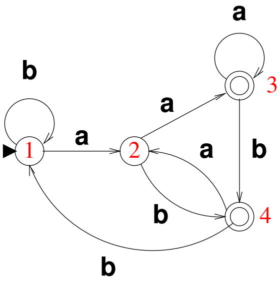
Determinism

(a | b)*a(a | b):

Nondeterministic:
(NFA)

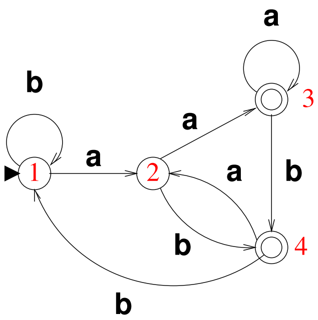


Deterministic:
(DFA)



Recognition with a DFA

Is abab $\in \mathcal{L}((\mathbf{a} \mid \mathbf{b})^*\mathbf{a}(\mathbf{a} \mid \mathbf{b}))$?



Input:		a	b	a	b	Accept?
Path:	1	2	4	2	4	YES

NFA vs. DFA

For every NFA, there is a DFA that accepts the same set of strings.

- NFA may have transitions labeled by ϵ.
  (Spontaneous transitions)
- All transition labels in a DFA belong to Σ.
- For some string x, there may be *many* accepting paths in an NFA.
- For all strings x, there is *one unique* accepting path in a DFA.
- Usually, an input string can be recognized *faster* with a DFA.
- NFAs are typically *smaller* than the corresponding DFAs.

NFA vs. DFA (contd.)

R = Size of Regular Expression
N = Length of Input String

	NFA	**DFA**
Size of Automaton	$O(R)$	$O(2^R)$
Recognition time per input string	$O(N \times R)$	$O(N)$

Converting NFA to DFA

Subset construction

Given a set S of NFA states,

- compute $S_\epsilon = \epsilon\text{-closure}(S)$: S_ϵ is the set of all NFA states reachable by zero or more ϵ-transitions from S.
- compute $S_\alpha = \text{goto}(S, \alpha)$:
  - S' is the set of all NFA states reachable from S by taking a transition labeled α.
  - $S_\alpha = \epsilon\text{-closure}(S')$.

Converting NFA to DFA (contd).

- Each state in DFA corresponds to a *set of states* in NFA.

Converting NFA to DFA (contd).

- Each state in DFA corresponds to a *set of states* in NFA.
- Start state of DFA $= \epsilon$-closure(start state of NFA).

Converting NFA to DFA (contd).

- Each state in DFA corresponds to a *set of states* in NFA.
- Start state of DFA $= \epsilon$-closure(start state of NFA).
- From a state s in DFA that corresponds to a set of states S in NFA:

Converting NFA to DFA (contd).

- Each state in DFA corresponds to a *set of states* in NFA.
- Start state of DFA = ϵ-closure(start state of NFA).
- From a state s in DFA that corresponds to a set of states S in NFA:
  - let $S' = \text{goto}(S, \alpha)$ such that S' is non-empty.

Converting NFA to DFA (contd).

- Each state in DFA corresponds to a *set of states* in NFA.
- Start state of DFA = ϵ-closure(start state of NFA).
- From a state s in DFA that corresponds to a set of states S in NFA:
  - let $S' = \text{goto}(S, \alpha)$ such that S' is non-empty.
  - add an α-transition to state s' that corresponds S' in NFA,

Converting NFA to DFA (contd).

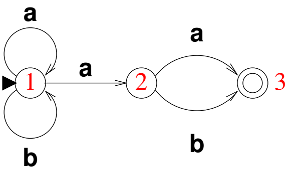
- Each state in DFA corresponds to a *set of states* in NFA.
- Start state of DFA = ϵ-closure(start state of NFA).
- From a state s in DFA that corresponds to a set of states S in NFA:
  - let $S' = \text{goto}(S, \alpha)$ such that S' is non-empty.
  - add an α-transition to state s' that corresponds S' in NFA,
- S contains a final NFA state, and s is the corresponding DFA state

Converting NFA to DFA (contd).

- Each state in DFA corresponds to a *set of states* in NFA.
- Start state of DFA = ϵ-closure(start state of NFA).
- From a state *s* in DFA that corresponds to a set of states *S* in NFA:
  - let $S' = \text{goto}(S, \alpha)$ such that S' is non-empty.
  - add an α-transition to state s' that corresponds S' in NFA,
- *S* contains a final NFA state, and *s* is the corresponding DFA state

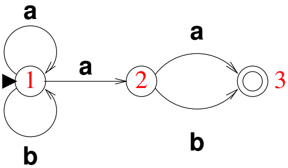
  $\Rightarrow$ *s* is a final state of DFA

NFA → DFA: An Example



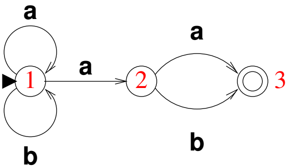
ϵ-closure($\{1\}$) $=$ $\mathbf{\{1\}}$

NFA $\rightarrow$ DFA: An Example



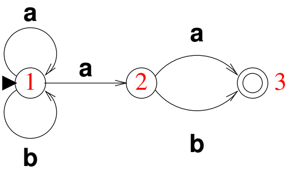
$$\begin{array}{lcl} \epsilon\text{-closure}(\{1\}) & = & \mathbf{\{1\}} \\ \text{goto}(\{1\}, \text{a}) & = & \mathbf{\{1, 2\}} \end{array}$$

NFA → DFA: An Example



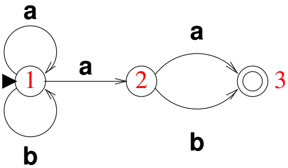
$$
\begin{aligned}
\epsilon\text{-closure}(\{1\}) &= \{1\} \\
\text{goto}(\{1\}, \text{a}) &= \{1, 2\} \\
\text{goto}(\{1\}, \text{b}) &= \{1\}
\end{aligned}
$$

NFA → DFA: An Example



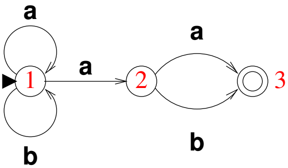
$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \mathbf{\{1\}} \\
\text{goto}(\{1\}, \mathtt{a}) & = & \mathbf{\{1, 2\}} \\
\text{goto}(\{1\}, \mathtt{b}) & = & \{1\} \\
\text{goto}(\{1, 2\}, \mathtt{a}) & = & \mathbf{\{1, 2, 3\}}
\end{array}
$$

NFA → DFA: An Example



$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \mathbf{\{1\}} \\
\text{goto}(\{1\}, \texttt{a}) & = & \mathbf{\{1, 2\}} \\
\text{goto}(\{1\}, \texttt{b}) & = & \{1\} \\
\text{goto}(\{1, 2\}, \texttt{a}) & = & \mathbf{\{1, 2, 3\}} \\
\text{goto}(\{1, 2\}, \texttt{b}) & = & \mathbf{\{1, 3\}}
\end{array}
$$

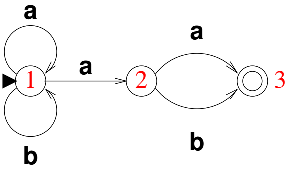
NFA → DFA: An Example



$$
\begin{aligned}
\epsilon\text{-closure}(\{1\}) &= \mathbf{\{1\}} \\
\text{goto}(\{1\}, \texttt{a}) &= \mathbf{\{1, 2\}} \\
\text{goto}(\{1\}, \texttt{b}) &= \{1\} \\
\text{goto}(\{1, 2\}, \texttt{a}) &= \mathbf{\{1, 2, 3\}} \\
\text{goto}(\{1, 2\}, \texttt{b}) &= \mathbf{\{1, 3\}}
\end{aligned}
$$

$$
\text{goto}(\{1, 2, 3\}, \texttt{a}) = \mathbf{\{1, 2, 3\}}
$$

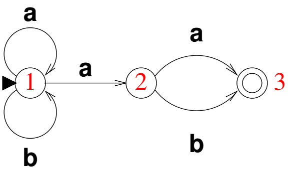
NFA → DFA: An Example



$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \mathbf{\{1\}} \\
\text{goto}(\{1\}, \mathtt{a}) & = & \mathbf{\{1, 2\}} \\
\text{goto}(\{1\}, \mathtt{b}) & = & \{1\} \\
\text{goto}(\{1, 2\}, \mathtt{a}) & = & \mathbf{\{1, 2, 3\}} \\
\text{goto}(\{1, 2\}, \mathtt{b}) & = & \mathbf{\{1, 3\}}
\end{array}
$$

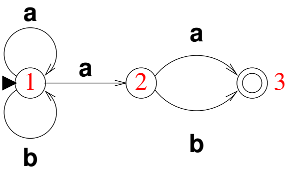
$$
\begin{array}{lcl}
\text{goto}(\{1, 2, 3\}, \mathtt{a}) & = & \mathbf{\{1, 2, 3\}} \\
\text{goto}(\{1, 2, 3\}, \mathtt{b}) & = & \{1, 3\}
\end{array}
$$

NFA → DFA: An Example



$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \mathbf{\{1\}} \\
\text{goto}(\{1\}, \mathtt{a}) & = & \mathbf{\{1,2\}} \\
\text{goto}(\{1\}, \mathtt{b}) & = & \{1\} \\
\text{goto}(\{1,2\}, \mathtt{a}) & = & \mathbf{\{1,2,3\}} \\
\text{goto}(\{1,2\}, \mathtt{b}) & = & \mathbf{\{1,3\}}
\end{array}
\qquad
\begin{array}{lcl}
\text{goto}(\{1,2,3\}, \mathtt{a}) & = & \mathbf{\{1,2,3\}} \\
\text{goto}(\{1,2,3\}, \mathtt{b}) & = & \{1,3\} \\
\text{goto}(\{1,3\}, \mathtt{a}) & = & \{1,2\}
\end{array}
$$

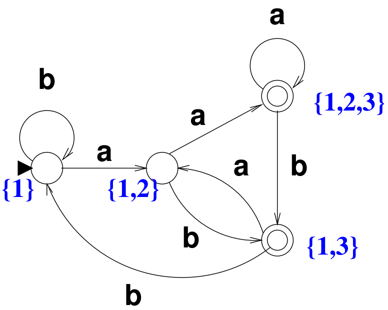
NFA → DFA: An Example



$$
\begin{array}{lcl}
\epsilon\text{-closure}(\{1\}) & = & \mathbf{\{1\}} \\
\text{goto}(\{1\}, \texttt{a}) & = & \mathbf{\{1,2\}} \\
\text{goto}(\{1\}, \texttt{b}) & = & \{1\} \\
\text{goto}(\{1,2\}, \texttt{a}) & = & \mathbf{\{1,2,3\}} \\
\text{goto}(\{1,2\}, \texttt{b}) & = & \mathbf{\{1,3\}}
\end{array}
\qquad
\begin{array}{lcl}
\text{goto}(\{1,2,3\}, \texttt{a}) & = & \mathbf{\{1,2,3\}} \\
\text{goto}(\{1,2,3\}, \texttt{b}) & = & \{1,3\} \\
\text{goto}(\{1,3\}, \texttt{a}) & = & \{1,2\} \\
\text{goto}(\{1,3\}, \texttt{b}) & = & \{1\}
\end{array}
$$

NFA → DFA: An Example (contd.)

ϵ-closure($\{1\}$)	$=$	$\{1\}$			
goto($\{1\}$, a)	$=$	$\{1, 2\}$	goto($\{1, 2, 3\}$, a)	$=$	$\{1, 2, 3\}$
goto($\{1\}$, b)	$=$	$\{1\}$	goto($\{1, 2, 3\}$, b)	$=$	$\{1, 3\}$
goto($\{1, 2\}$, a)	$=$	$\{1, 2, 3\}$	goto($\{1, 3\}$, a)	$=$	$\{1, 2\}$
goto($\{1, 2\}$, b)	$=$	$\{1, 3\}$	goto($\{1, 3\}$, b)	$=$	$\{1\}$



Construction of a Lexical Analyzer

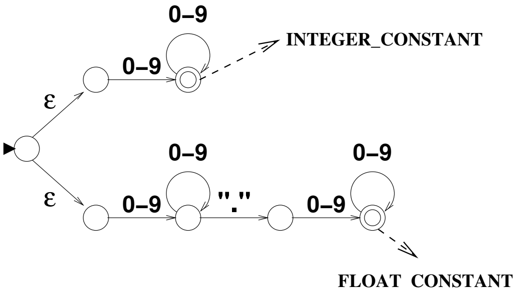
- Regular Expressions and Definitions are used to specify the set of strings (lexemes) corresponding to a *token*.
- An automaton (DFA/NFA) is built from the above specifications.
- Each final state is associated with an *action*: emit the corresponding token.

Specifying Lexical Analysis

Consider a recognizer for integers (sequence of digits) and floats (sequence of digits separated by a decimal point).

```
[0-9]+                { emit(INTEGER_CONSTANT); }

[0-9]+"."[0-9]+       { emit(FLOAT_CONSTANT); }
```



Lex

- Tool for building lexical analyzers.
- Input: lexical specifications (`.l` file)
- Output: C function (`yylex`) that returns a token on each invocation.
- Example:

```
%%
[0-9]+                  { return(INTEGER_CONSTANT); }

[0-9]+"."[0-9]+         { return(FLOAT_CONSTANT); }
```

- Tokens are simply integers (`#define`'s).

Lex Specifications

```
%{
    C header statements for inclusion
%}
  Regular Definitions    e.g.:
    digit    [0-9]
%%

  Token Specifications    e.g.:
    {digit}+                  { return(INTEGER_CONSTANT); }

%%
  Support functions in C
```

Lex/Flex Regular Expressions

Adds "syntactic sugar" to regular expressions:

- Range: `[0-7]`: Integers from 0 through 7 (inclusive)
  `[a-nx-zA-Q]`: Letters `a` thru `n`, `x` thru `z` and `A` thru `Q`.
- Exception: `[^/]`: Any character other than `/`.
- Definition: `{digit}`: Use the previously specified regular definition `digit`.
- Special characters: Connectives of regular expression, convenience features.
  e.g.: `| * ^`

Special Characters in Lex/Flex

`| * + ? ( )`	Same as in regular expressions
`[ ]`	Enclose ranges and exceptions
`{ }`	Enclose "names" of regular definitions
`^`	Used to negate a specified range (in Exception)
`.`	Match any single character except newline
`\`	Escape the next character
`\n`, `\t`	Newline and Tab

For literal matching, enclose special characters in double quotes (`"`) *e.g.:* `"*"`

Or use "\" to escape. *e.g.:* `\*`

Examples

`for`	Sequence of `f`, `o`, `r`

Examples

`for`	Sequence of `f`, `o`, `r`
`"||"`	C-style OR operator (two vert. bars)

Examples

`for`	Sequence of `f`, `o`, `r`
`"||"`	C-style OR operator (two vert. bars)
`.*`	Sequence of non-newline characters

Examples

<table>
<tr><td>for</td><td>Sequence of f, o, r</td></tr>
<tr><td>"||"</td><td>C-style OR operator (two vert. bars)</td></tr>
<tr><td>.*</td><td>Sequence of non-newline characters</td></tr>
<tr><td>[^*/]+</td><td>Sequence of characters except * and /</td></tr>
</table>

Examples

`for`	Sequence of `f`, `o`, `r`
`"||"`	C-style OR operator (two vert. bars)
`.*`	Sequence of non-newline characters
`[^*/]+`	Sequence of characters except `*` and `/`
`\"[^"]*\"`	Sequence of non-quote characters beginning and ending with a quote

Examples

<table>
<tr><td><code>for</code></td><td>Sequence of <code>f</code>, <code>o</code>, <code>r</code></td></tr>
<tr><td><code>"||"</code></td><td>C-style OR operator (two vert. bars)</td></tr>
<tr><td><code>.*</code></td><td>Sequence of non-newline characters</td></tr>
<tr><td><code>[^*/]+</code></td><td>Sequence of characters except <code>*</code> and <code>/</code></td></tr>
<tr><td><code>\"[^"]*\"</code></td><td>Sequence of non-quote characters
beginning and ending with a quote</td></tr>
<tr><td><code>({letter}|"_")({letter}|{digit}|"_")*</code></td><td>C-style identifiers</td></tr>
</table>

Actions

Actions are attached to final states.
Actions:

- Distinguish the different final states.
- Are used to return *tokens*.
- Can be used to set *attribute values*.
- In Lex/Flex: action is a fragment of C code (blocks enclosed by '{' and '}').

PLY

PLY is a Yacc/Lex-like parser/lexer framework in Python.
(See `http://www.dabeaz.com/ply/`)

- List of tokens is declared *a priori*.
- Each token T with an action is specified by a Python function `t_`T:
  - Regular expression patterns, specified as Python docstrings, describe sets of lexemes.
  - Function body describes the action to be performed when input matches the pattern.
- Action-less token T is specified by defining variable `t_`T with the regular expression pattern as its value.

Example:

```
import ply.lex as lex

# List of token names.
tokens = ('NUMBER', 'PLUS',
   'MINUS')

# Tokens without actions.
t_PLUS = r'\+'
t_MINUS = r'-'

# A token with action.
def t_NUMBER(t):
    r'\d+'
    t.value = int(t.value)
    return t
```

Mechanics

- The returned tokens are instances of class `LexToken`, with attributes `type`, `value`, `lineno` and `lexpos`.
- Line numbers have to be maintained explicitly by setting the `lineno` attribute of the lexer.
- To ignore a lexeme (i.e. not return a token),
  - end its action with a `pass` instead of `return`, or
  - name the rule as `t_ignore`
- Error handling (for characters not matching any pattern) can be specified as function `t_error`.
- PLY's lexer can handle regular definitions, as well as conditional analysis (*a la* lex) where matching can be controlled by explicitly maintained conditions. See PLY documentation for details.

Priority of matching

- Patterns for tokens with actions are matched in the order they are specified.
- Regular expressions for action-less tokens are sorted, and matched longest-expression first.

Constructing Lexers using PLY

- Easy way:
  - `lex.lex()` to create the lexer;
  - `lex.input()` to specify the input string to be scanned;
  - repeated invocation of `lex.token()` to generate tokens.
- Alternative (better) way:
  - Put lexer specifications in a separate module, say `proto2lex.py`.
  - `lexer = lex.lex(module=proto2lex)` to create a lexer (referenced from variable `lexer`).
  - `lexer.input(...)` to specify its input
  - `lexer.token()` to generate tokens, one at a time.
- The alternative way works even when there are multiple instances of the same lexer in an application.

Lexical Analysis: Summary

Convert a stream of characters into a stream of tokens.

- Make rest of compiler independent of character set
- Strip off comments
- Recognize line numbers
- Ignore white space characters
- Process macros (definitions and uses)
- Interface with **symbol table** (also called "name table").