CS 320
Computer Language Processing
Exercises: Weeks 1 and 2

February 28, 2025

1 Languages and Automata

Exercise 1 Consider the following languages defined by regular expressions:

1. $\{a, ab\}^*$
2. $\{aa\}^* \cup \{aaa\}^*$
3. a^+b^+

and the following languages defined in set-builder notation:

A. $\{w \mid \forall i. 0 \leq i \leq |w| \land w_{(i)} = b \implies (i > 0 \land w_{(i-1)} = a)\}$

B. $\{w \mid \forall i. 0 \leq i < |w| - 1 \implies w_{(i)} = b \implies w_{(i+1)} = a\}$

C. $\{w \mid \exists i. 0 < i < |w| \land w_{(i)} = b \land w_{(i-1)} = a\}$

D. $\{w \mid (|w| = 0 \mod 2 \lor |w| = 0 \mod 3) \land \forall i. 0 \leq i < |w| \implies w_{(i)} = a\}$

E. $\{w \mid \forall i. 0 \leq i < |w| - 1 \land w_{(i)} = a \implies w_{(i+1)} = b\}$

F. $\{w \mid \exists i. 0 < i < |w| - 1 \land (\forall y. 0 \leq y \leq i \implies w_{(y)} = a) \land (\forall y. i < y < |w| \implies w_{(y)} = b)\}$

For each pair (e.g. 1-A), check whether the two languages are equal, providing a proof if they are, and a counterexample word that is in one but not the other if unequal.

Solution Equal language pairs: $1 \mapsto A, 2 \mapsto D, 3 \mapsto F$.

Counterexamples ($\cdot^\star$ means the word is in the alphabet-labelled language, and the number-labelled language otherwise):

	A	B	C	D	E	F
1	-	a	a	a	aa	a
2	ab*	ba*	ab*	-	ab*	aa
3	abb	abb	aba*	aaabb	aab	-

We prove the first case as an example.

$$\{a, ab\}^* = \{w \mid \forall i.0 \leq i \leq |w| \land w_{(i)} = b \implies (i > 0 \land w_{(i-1)} = a)\}$$

We must prove both directions, i.e. that $\{a, ab\}^* \subseteq \{w \mid P(w)\}$ and that $\{w \mid P(w)\} \subseteq \{a, ab\}^*$.

Forward: $\{a, ab\}^* \subseteq \{w \mid P(w)\}$:

We must show that for all $w \in \{a, ab\}^*$, $P(w)$ holds. For any $i \in \mathbb{N}$, given that $0 \leq i \leq |w| \land w_{(i)} = b$, we need to show that $i > 0 \land w_{(i-1)} = a$.

From the definition of $*$ on sets of words, we know that there must exist $n < |w|$ words $w_1, \ldots, w_n \in \{a, ab\}$ such that $w = w_1 \ldots w_n$. The index i must be in the range of one of these words, i.e. there exist $1 \leq m \leq n$ and $0 \leq j < |w_m|$ such that $w_{(i)} = w_{m(j)}$.

We know that $w_{(i)} = b$ and $w_m \in \{a, ab\}$ by assumption. The case $w_m = a$ is a contradiction, since it cannot contain b. Thus, $w_m = ab$. We know that $w_{(i)} = w_{m(j)} = b$, so $j = 1$. Thus, $w_{(i-1)} = w_{m(j-1)} = w_{m(0)} = a$, as required. Since $i - 1 \geq 0$, being an index into w, $i > 0$ holds as well. Hence, $P(w)$ holds.

Backward: $\{w \mid P(w)\} \subseteq \{a, ab\}^*$:

We must show that for all w such that $P(w)$ holds, $w \in \{a, ab\}^*$. We know by definition of $*$ again, that $w \in \{a, ab\}*$ if and only if there exist $n < |w|$ words $w_1, \ldots, w_n \in \{a, ab\}$ such that $w = w_1 \ldots w_n$. We attempt to show that if $P(w)$ holds, then w admits such a decomposition.

We proceed by induction on the length of w.

Induction Case $|w| = 0$: The empty word has a decomposition $w = \epsilon$ (with $n = 0$ in the decomposition). QED.

Induction Case $|w| = 1$: The word w is either a or b. We know that $P(w)$ holds, so $w = a$ (why?). The decomposition is $w = a$, with $n = 1$ and $w_1 = a$. QED.

Induction Case $|w| > 1$:

Induction hypothesis: for all words v such that $|v| < |w|$ and $P(v)$ holds, v admits a decomposition into words in $\{a, ab\}$, and thus $v \in \{a, ab\}^*$.

We need to show that if $P(w)$ holds, then w admits such a decomposition as well. Split the proof based on the first two characters of w. There are four possibilities. We give the name v to the rest of w.

1. $w = aav$: $P(w)$ holds, so $\forall i.0 \leq i \leq |w| \land w_{(i)} = b \implies (i > 0 \land w_{(i-1)} = a)$. In particular, we can restrict to $i > 1$ as

   $$\forall i.2 \leq i \leq |w| \land w_{(i)} = b \implies (i > 0 \land w_{(i-1)} = a)$$

   but $w_{(i)}$ for $i \geq 2$ is simply $v_{(i-2)}$. Rewriting:

   $$\forall i.2 \leq i \leq |w| \land v_{(i-2)} = b \implies (i > 0 \land v_{(i-3)} = a)$$

   Finally, since the statement holds for all i, we can replace i by $i+2$ without loss of generality, using $|v| = |w| - 2$:

   $$\forall i.0 \leq i \leq |v| \land v_{(i)} = b \implies (i > 0 \land v_{(i-1)} = a)$$

This is precisely the statement $P(v)$, so by the induction hypothesis, v has a decomposition into words in $\{a, ab\}$, $v = v_1 \dots v_m$ for some $m < |v|$ and $v_i \in \{a, ab\}$.

We can now construct a decomposition for w, $w = w_1 \dots w_{m+2}$ such that $w_1 = a$, $w_2 = a$, and $w_{i+2} = v_i$ for $1 \le i \le m$. Since $m < |v|$ and $|v| = |w| - 2$, $m + 2 < |w|$. QED.

2. $w = aab$: by the same argument as the previous case, v has a decomposition into words in $\{a, ab\}$, $v = v_1 \dots v_m$ for some $m < |v|$ and $v_i \in \{a, ab\}$.

   We can similarly construct a decomposition for w, $w = w_1 \dots w_{m+1}$ such that $w_1 = ab$ and $w_{i+1} = v_i$ for $1 \le i \le m$. Since $m < |v|$ and $|v| = |w| - 2$, in particular $m + 1 < |w|$. QED.

3. $w = bav$ or $w = bbv$: $P(w)$ cannot hold (set $i = 0$), so the statement is vacuously true.

□

Exercise 2 For each the following languages, construct an NFA $\mathcal{A}$ that recognizes them, i.e. $L(\mathcal{A}) = L_i$:

1. L_1: binary strings divisible by 3
2. L_2: binary strings divisible by 4
3. L_3: $\{(w_1 \oplus w_2) \mid w_1 \in L_1 \wedge w_2 \in L_2 \wedge |w_1| = |w_2|\}$

where $\oplus$ is the bitwise-xor operation on binary strings.

Solution

1. The language of binary strings divisible by 3. We need two observations to construct this automaton:

   (a) If the automaton has consumed a binary string s with decimal value, say, $val(s) = n$, then we can determine the decimal value of the string after reading one more character as either $val(s0) = 2n$ or $val(s1) = 2n + 1$.

   (b) The set of strings is finite, but it is sufficient to know only the value of the string *modulo 3* to determine if it is divisible.

   We construct the automaton $\mathcal{A}_1 = (Q, \Sigma, \delta, q_{init}, F)$ where:

   - $Q = \{q_{init}, q_0, q_1, q_2\}$, representing the initial state (empty word has no value), and the states corresponding to the values $0, 1, 2$ modulo 3.
   - $\Sigma = \{0, 1\}$, as required.
   - $\delta = \{(q_i, 0, q_j) \mid 2i \mod 3 = j\} \cup \{(q_i, 1, q_j) \mid (2i + 1) mod 3 = j\} \cup \{(q_{init}, 0, q_0), (q_{init}, 1, q_1)\}$, i.e., there is a transition from q_i to q_j if, as the currently known value modulo 3 is i, on reading 0 the next value is $j = 2i \mod 3$. We use the fact that $2n \mod 3 = 2(n \mod 3) \mod 3$. The case for reading 1 is similar. The translations from the

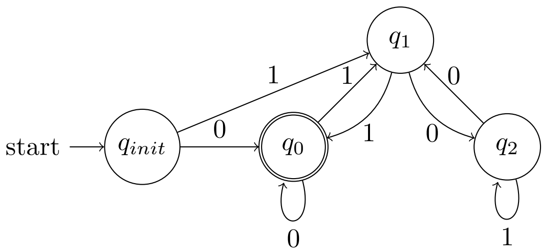
initial state are to the states corresponding to the values $0, 1$ modulo 3.

For example, if we have read "1101", with decimal value 13, we must be in state q_1, as $13 \mod 3 = 1$. On reading a 0, we have the string "11010" with decimal value 26, and $26 \mod 3 = 2$, so we transition to q_2.

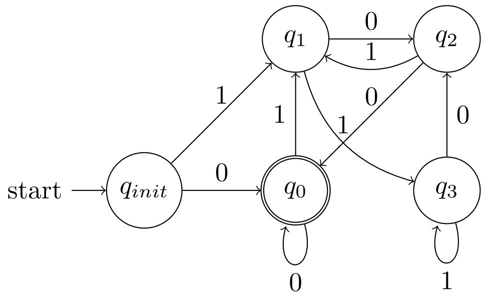
The full automaton is below.

- $F = \{q_0\}$ as we accept that words that are divisible by 3, and are hence equal to 0 modulo 3.

The automaton is:



2. The language of binary strings divisible by 4. The construction is similar to the one above, now with 5 states.



3. To compute the bitwise-xor of two strings, we must compute a product automaton. To accept a word w, there must exist w_1, w_2 such that $w_1 \in L_1$, and $w_2 \in L_2$.

   We do not explicitly construct the automaton, but present an argument. First, consider the truth table for xor:

b_1	b_2	$b_1 \oplus b_2$
0	0	0
0	1	1
1	0	1
1	1	0

   Notably, given a xor result, we cannot exactly determine the input bits. In essence, we construct an automaton that, given a string, tries to simulate the two input automata in parallel non-deterministically on all possible pairs of input strings. If any of them are accepted, that means we found a pair of strings that, one, are accepted by the two original automata, and two, have the input string as their bitwise-xor.

Formally, the automaton $\mathcal{A}_3 = (Q, \Sigma, \delta, q_{init}, F)$ has:

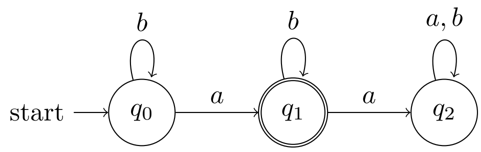
- $Q = Q_1 \times Q_2$, where Q_1 and Q_2 are the state sets of $\mathcal{A}_1$ and $\mathcal{A}_2$.
- $\Sigma = \{0, 1\}$ as before.
- $q_{init} = (q_{1,init}, q_{2,init})$ where $q_{1,init}$ and $q_{2,init}$ are the initial states of $\mathcal{A}_1$ and $\mathcal{A}_2$.
- $F = F_1 \times F_2$ similarly.
- δ is constructed as follows: for a pair of states (q_1, q_2), on reading a 0, we look at the truth table of xor; two input pairs $(0, 0)$ and $(1, 1)$ could have produced this result bit. Hence, we add transitions for both automata simultaneously, $(((q_1, q_2), 0, (q_1', q_2')))$ corresponding to possible inputs $(0, 0)$ if $\delta_1(q_1, 0, q_1')$ and $\delta_2(q_2, 0, q_2')$, and similarly $((q_1, q_2), 0, (q_1', q_2'))$ corresponding to possible inputs $(1, 1)$ if $\delta_1(q_1, 1, q_1'')$ and $\delta_2(q_2, 1, q_2'')$.

  The case for reading a 1 is similar, with possible input pairs $(0, 1)$ and $(1, 0)$.

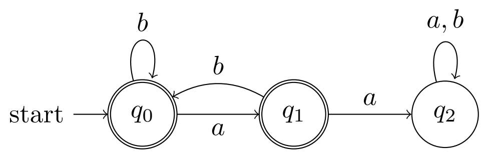
$\square$

Exercise 3 Give a verbal and a set-notational description of the language accepted by each of the following automata. You can assume that the alphabet is $\Sigma = \{a, b\}$.

1. $\mathcal{A}_1$



2. $\mathcal{A}_2$



Solution

1. As regular expression: b^*ab^*, this is the language of words that contain exactly one a. In set-notation:

$$\{w \mid \exists! i.\ 0 \leq i \leq |w| \wedge w_{(i)} = a\}$$

2. As generalized regular expression (with complement): $(\Sigma^* aa \Sigma^*)^c$. Without complement: $b^*(ab^+)^*(a \mid \epsilon)$. This is the language of words that contain no consecutive pair of a's. In set-notation:

$$\{w \mid \forall i.\ 0 \leq i < |w| \land w_{(i)} = a \implies (i + 1 \geq |w| \lor w_{(i+1)} \neq a)\}$$

$\square$

2 Lexing

Consider a simple arithmetic language that allows you to compute one arithmetic expression, construct conditionals, and let-bind expressions. An example program is:

```
let x = 3 in
let y = ite (x > 0) (x * x) 0 in
  (2 * x) + y
```

The lexer for this language must recognize the following tokens:

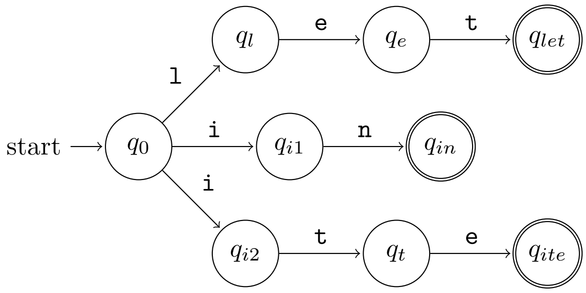
$$\begin{aligned}
\texttt{keyword} &: \ \texttt{let} \mid \texttt{in} \mid \texttt{ite} \\
\texttt{op} &: \ \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \\
\texttt{comp} &: \ \texttt{>} \mid \texttt{<} \mid \texttt{==} \mid \texttt{<=} \mid \texttt{>=} \\
\texttt{equal} &: \ \texttt{=} \\
\texttt{lparen} &: \ \texttt{(} \\
\texttt{rparen} &: \ \texttt{)} \\
\texttt{id} &: \ \mathit{letter} \cdot (\mathit{letter} \mid \mathit{digit})^* \\
\texttt{number} &: \ \mathit{digit}^+ \\
\texttt{skip} &: \ \texttt{whitespace}
\end{aligned}$$

For simplicity, *letter* is a shorthand for the set of all English lowercase letters $\{a - z\}$ and *digit* is a shorthand for the set of all decimal digits $\{0 - 9\}$.

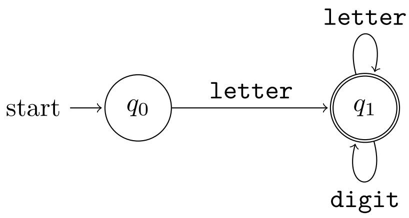
Exercise 4 For each of the tokens above, construct an NFA that recognizes strings matching its regular expression.

Solution The construction is similar in each case, following translation of regular expressions to automata. For example:

- `keyword`: `let` | `in` | `ite`



- `id`: `letter` · (`letter` | `digit`)*



The other cases are similar. □

A lexer is constructed by combining the NFAs for each of the tokens in parallel, assuming maximum munch. The resulting token is the first NFA in the token order that accepts a prefix of the string. Thus, tokens listed first have higher priority. We then continue lexing the remaining string. You may assume that the lexer drops any `skip` tokens.

Exercise 5 For each of the following strings, write down the sequence of tokens that would be produced by the constructed lexer, if it succeeds.

1. `let x = 5 in x + 3`
2. `let5x2`
3. `xin`
4. `==>`
5. `<===><==`

Solution

1. `[keyword("let"), id("x"), equal("="), number("5"), keyword("in"), id("x"), op("+"), number("3")]`
2. `[keyword("let"), number("5"), id("x2")]`
3. `[id("xin")]`
4. `[comp("=="), comp(">")]`
5. `[comp("<="), comp("=="), comp(">"), comp("<="), equal("=")]`

□

Exercise 6 Construct a string that would be lexed differently if we ran the NFAs in parallel and instead of using token priority, simply picked the longest match.

Solution There are many possible solutions. The key is to notice which tokens have overlapping prefixes.

An example is `letx1`, which would be lexed as `[keyword("let"), id("x1")]` if we check acceptance in order of priority, but as `[id("letx1")]` if we run them in parallel. □