Lexical Analysis

Robert Chen

August 16, 2021

Contents

1 Regular Expressions

The following is the definition of a *regular expression*:

Definition. A *regular expression* (*regex* for short) over finite alphabet Σ is recursively built from the following rules:

- Any $a \in \Sigma$ is a regex.
- If α, β are regex, then $\alpha\beta$ is a regex.
- If α, β are regex, then $\alpha|\beta$ is a regex.
- If α is a regex, then α^* is a regex.

A regex defines a language — a set of strings it accepts. The following is the definiion of the language of a regex:

Definition. Let γ be a regex over Σ and $L(\gamma)$ denote the language it accepts, then

- If $\gamma = \varnothing$, then
$$L(\gamma) = \varnothing$$
- If $\gamma = a \in \Sigma$, then
$$L(\gamma) = \{a\}$$
- If $\gamma = \alpha\beta$, then
$$L(\gamma) = \{w_1 \parallel w_2 : w_1 \in L(\alpha), w_2 \in L(\beta)\}$$
where $\parallel$ denotes the concatenation operation on two strings.
- If $\gamma = \alpha|\beta$, then
$$L(\gamma) = L(\alpha) \cup L(\beta)$$
- If $\gamma = \alpha^*$, then
$$L(\gamma) = \bigcup_{w \in L(\alpha)} \bigcup_{n \in \mathbb{N}} w^n$$
where w^n denotes n concatenations of the same string w. Note that w^0 denotes the empty string (string of length 0), and is commonly denoted as ε.

We use regexs as a tool to describe what each type of token should look like, and the language it defines a set containing all valid tokens of that type. For common use, regexes are often defined over the ASCII characters and extended with the following:

- $x_a - x_b \equiv x_a|x_{a+1}|\cdots|x_b$: a range (can only be used in character classes), where x_i represents the i-th ASCII character.
- $[a_1a_2\cdots a_n] \equiv a_1|a_2|\cdots|a_n$: a character class, where each a_i is either from Σ or a range.
- $[\hat{}x] \equiv [a_1a_2\cdots a_n]$: a negated character class, where $\{a_1, a_2, \ldots, a_n\} = \Sigma \setminus L([x])$.
- $. = [x_0 - x_{127}]$: match any ASCII character.
- $\backslash d \equiv [0-9]$: digits.
- $\backslash w \equiv [A - Za - z]$: letters.
- $\alpha? \equiv \alpha|\varnothing^*$: 0 or 1 occurrences of α.
- $\alpha+ \equiv \alpha\alpha^*$: 1 or more occurrences of α.

2 Non-deterministic Finite Automata

A *non-deterministic finite automata* is defined as the following:

Definition. A *non-deterministic finite automata* (*NFA* for short) is a 5-tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where

- Σ is a finite alphabet.
- Q is a set of states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is a set of accepting states.
- $\delta \subseteq (Q \times (\Sigma \cup \{\varepsilon\})) \times Q$ denotes a set of transitions. Let $\delta(p, a) = q$ denote every $((p, a), q) \in \delta$.

An NFA also defines a language:

Definition. Let $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ be an NFA and $L(\mathcal{A})$ denote the language it accepts, then

$$L(\mathcal{A}) = \{w \in \Sigma^* : \exists q_0, q_1, \ldots, q_n \in Q \quad \alpha(a_1, a_2, \ldots, a_n, w) \wedge \beta(q_0, a_1, q_1, a_2, q_2, \ldots, a_n, q_n, \delta) \wedge (q_n \in F)\}$$

where

- Σ^* denotes all strings over Σ. In other words,
$$\Sigma^* = \bigcup_{n \in \mathbb{N}} \{a_1 \parallel a_2 \parallel \cdots \parallel a_n : a_1, a_2, \ldots, a_n \in \Sigma\}$$
- $\alpha(a_1, a_2, \ldots, a_n, w)$ denotes the following boolean expression:
$$a_1 \parallel a_2 \parallel \cdots \parallel a_n = w$$
- $\beta(q_0, a_1, q_1, a_2, q_2, \ldots, a_n, q_n, \delta)$ defines the following boolean expression:
$$\bigwedge_{i=0,\ldots,n-1} \delta(q_i, a_{i+1}) = q_{i+1}$$

The connection between NFA and regex is the following theorem:

Theorem. *For every regex γ over finite alphabet Σ, there is a NFA $\mathcal{A}$ such that $L(\gamma) = L(\mathcal{A})$.*

Proof. We prove this by induction on γ:

- Base case:
  - If $\gamma = \varnothing$, then let $\mathcal{A} = (\Sigma, \{q_0\}, q_0, \varnothing, \varnothing)$.
  - If $\gamma = a \in \Sigma$, then let $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where
    * $Q = \{q_0, q_1\}$.
    * $F = \{q_1\}$.
    * $\delta = \{((q_0, a), q_1)\}$.

  In both cases, it is obvious that $L(\alpha) = \{a\} = L(\mathcal{A})$.

- Induction case: suppose α, β are regexes, $\mathcal{A}_1 = (\Sigma, Q_1, q_{10}, F_1, \delta_1), \mathcal{A}_2 = (\Sigma, Q_2, q_{20}, F_2, \delta_2)$ are NFAs, and $L(\alpha) = L(\mathcal{A}_1), L(\beta) = L(\mathcal{A}_2)$ by inductive hypothesis, then
  - If $\gamma = \alpha\beta$, then let $\mathcal{A} = (\Sigma, Q_1 \cup Q_2, q_{10}, F_2, \delta)$ where
  $$\delta = \delta_1 \cup \delta_2 \cup ((F_1 \times \{\varepsilon\}) \times \{q_{20}\})$$
  - If $\gamma = \alpha|\beta$, then let $\mathcal{A} = (\Sigma, Q_1 \cup Q_2 \cup \{q_0\}, q_0, F_1 \cup F_2, \delta)$ where
  $$\delta = \delta_1 \cup \delta_2 \cup ((\{q_0\} \times \{\varepsilon\}) \times \{q_{10}, q_{20}\})$$

– If $\gamma = \alpha^*$, then let $\mathcal{A} = (\Sigma, Q_1 \cup \{q_0\}, q_0, F_1 \cup \{q_0\}, \delta)$ where

$$\delta = \delta_1 \cup ((F_1 \times \{\varepsilon\}) \times \{q_0\}) \cup \{((q_0, \varepsilon), q_{10})\}$$

In all three cases, it is clear that $L(\gamma) = L(\mathcal{A})$.

□

This proof also gives a (recursive) constructive algorithm for building an NFA that accepts the same language as an arbitrary regex.

3 Deterministic Finite Automata

An NFA is not ideal for implementation because:

- From a state q reading input a, we can go to multiple different states.
- From a state q without reading any input, we can go to multiple different states. These are called ε-transitions.

A *deterministic finite automata* solves this issue:

Definition. A *deterministic finite automata* (*DFA* for short) is a 5-tuple $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ where

- Σ is a finite alphabet.
- Q is a set of states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is a set of accepting states.
- $\delta : I \to Q$ is a set of transitions where $I \subseteq Q \times \Sigma$.

The only difference between a DFA and NFA is the set of transitions. The definition of the language accepted by a DFA is also the same as that of an NFA. The connection between a DFA and an NFA is the following theorem:

Theorem. *For every NFA $\mathcal{A}$, there is a DFA $\mathcal{A}'$ such that $L(\mathcal{A}) = L(\mathcal{A}')$.*

We omit the proof, and only show how to construct such DFA. We first define what an *ε-closure* is:

Definition. Let $\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$ be an NFA, and let $q \in Q$. The *ε-closure of q* is a set

$$\{p \in Q : \exists q_1, q_2, \ldots, q_n \in Q \quad (q_1 = q) \wedge (q_n = p) \wedge \alpha(q_1, q_2, \ldots, q_n)\}$$

where

$$\alpha(q_1, q_2, \ldots, q_n) = \bigwedge_{i=1}^{n-1} \delta(q_i, \varepsilon) = q_{i+1}$$

We typically denote such a set as ε-closure(q).

Then, the following is the algorithm to construct the DFA $\mathcal{A}'$ from an NFA $\mathcal{A}$:

Algorithm 1 NFA To DFA Conversion

function NFA-To-DFA-Conversion($\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$)
  $Q' \leftarrow \varnothing$
  for $q \in Q$ **do**
    $Q' \leftarrow Q' \cup \{\varepsilon\text{-closure}(q)\}$ ▷ each state of the DFA is a subset of Q
  end for
  $q'_0 \leftarrow \varepsilon\text{-closure}(q_0)$
  $F' \leftarrow \varnothing$
  for $q' \in Q'$ **do**
    if $q' \cap F \neq \varnothing$ **then**

$F' \leftarrow F' \cup \{q'\}$ ▷ any DFA state containing an accepting NFA state is an accepting DFA state
end if
end for
$\delta' \leftarrow \varnothing$
for $p' \in Q'$ **do**
for $a \in \Sigma$ **do**
$q' \leftarrow \varnothing$
for $p \in p'$ **do**
for $((p, a), q) \in \delta$ **do**
$q' \leftarrow q' \cup \varepsilon\text{-closure}(q)$ ▷ combine all reachable NFA states (when going from an NFA state $p \in p'$ and reading a) into a DFA state q'
end for
end for
$Q' \leftarrow Q' \cup \{q'\}$
$\delta' \leftarrow \delta' \cup \{((p', a), q')\}$
end for
end for
$\mathcal{A}' \leftarrow (\Sigma, Q', q_0', F', \delta')$
return $\mathcal{A}'$
end function

4 DFA Minimization

For efficent lexing, we would want the minimal DFA that accepts the same language. The following theorem applies to all DFA:

Theorem. *For every DFA $\mathcal{A}$, there is a unique minimal DFA $\mathcal{A}' = (\Sigma, Q', q_0', F', \delta')$ such that*

- *$L(\mathcal{A}) = L(\mathcal{A}')$.*
- *$|Q'|$ is minimized.*

This proof is also omitted. We provide the algorithm to minimize such DFA:

Algorithm 2 DFA Minimization

function DFA-Minimization($\mathcal{A} = (\Sigma, Q, q_0, F, \delta)$)
$P \leftarrow \{F, Q \setminus F\}$ ▷ partition the original states Q, each partition represents a set of potentially indistinguishable states
changed $\leftarrow$ **true**
while changed **do**
changed $\leftarrow$ **false**
for $p' \in P$ **do**
comesFrom $\leftarrow$ an empty dictionary
for $a \in \Sigma$ **do**
for $p \in p'$ **do**
$q' \leftarrow q' \in P$ such that $\delta(p, a) \in q'$ ▷ check which partition q' each state p in the current partition p' goes to when reading a
if comesFrom[q'] exists **then**
comesFrom[q'] $\leftarrow$ comesFrom[q'] $\cup \{p\}$
else
comesFrom[q'] $\leftarrow \{p\}$
end if
end for
if $|$comesFrom.keys$| \neq 1$ **then** ▷ states in current partition P' goes to different partitions, thus we must change the current partition
$P \leftarrow (P \setminus \{p'\}) \cup$ comesFrom.values
changed $\leftarrow$ **true**
break

```
            end if
        end for
        if changed then
            break
        end if
    end for
end while
Q′ ← ∅
F′ ← ∅
for p′ ∈ P do
    if p′ = ∅ then
        continue
    end if
    if q_0 ∈ p′ then
        p ← q_0                          ▷ make sure to choose q_0
    else
        p ← any p ∈ p′                   ▷ otherwise, choosing any state from the partition p′ is suffi-
                                           cient because they are indistinguishable
    end if
    Q′ ← Q′ ∪ {p}
    if p ∈ F then
        F′ ← F′ ∪ {p}
    end if
end for
δ′ ← ∅
for ((p, a), q) ∈ δ do
    if p ∈ Q′ and q ∈ Q′ then
        δ′ ← δ′ ∪ {((p, a), q)}
    end if
end for
𝒜 ← (Σ, Q′, q_0, F′, δ′)
return 𝒜
end function
```

5 Multiple Tokens

So far we have only considered building a DFA for one token. To build a DFA for multiple tokens, we do the following steps:

1. Build the regex for each token.
2. Convert the regexes to NFAs.
3. Combine the NFAs into one big NFA using ε-transitions.
4. Convert the big NFA into a DFA.
5. Minimize the DFA.

The only tricky part is minimizing the DFA. To be able to identify which type of token is read, we must initially split the accepting states of the DFA into different partitions if the accepting states it contains from different the original NFAs. Furthermore, because each accepting state in the DFA can now contain accepting states from different NFAs, choosing when to accept and which kind of token to accept will depend on

- Longest matching rule: match the longest token; this will require backtracking.
- User preference: define a priority on which tokens to be accepted first.
- Whitespace: typically, a programming language contains separating characters such as whitespace, and reading such character tells us to accept now.

6 Tokens Defined By Fartlang