CS241 Part II

Foundations of Sequential Programs

David Duan, 2019 Winter

Contents

1 Formal Languages

An **alphabet** is a non-empty finite set of symbols often denoted by Σ.

- English alphabet: $\Sigma = \{a, b, c, \ldots, z\}$.
- Binary strings: $\Sigma = \{0, 1\}$.
- Base-10 numbers: $\Sigma = \{0, 1, 2, \ldots, 9\}$.

A **string** (or **word**) w is a finite sequence of symbols chosen from Σ. The **length of a string** is denoted by $|w|$. The **set of all strings** over an alphabet Σ is denoted by Σ^*.

- Empty string: ε where $|\varepsilon| = 0$. Note that $\varepsilon \in \Sigma^*$ for all alphabet Σ.
- Binary string: $x = 01001$ where $|x| = 5$.

A **language** L is a set of strings.

- $L = \varnothing$ or $L = \{\}$: the empty language which does not contain any string.
- $L = \{\varepsilon\}$: the language consisting of the empty string.
- $L = \{ab^n a : n \in \mathbb{N}_0\}$: the set of strings over the alphabet $\Sigma = \{a, b\}$ consisting of an a followed by 0 or more b characters followed by an a.

Given a language, we want to determine if a string belongs to it.

2 Regular Languages

A **regular language** over an alphabet Σ consists of one of the following:

1. *Base Case:* The empty language ($\varnothing$) and the language consisting of the empty words $\{\varepsilon\}$ are regular.
2. *Base Case II:* All singleton (finite) languages are regular.
3. *Induction:* The **union**, **concatenation** or **Kleene star** (klay-nee) of any two regular languages are regular.
    a. *Union:* $(L := L_1 \cup L_2 = \{\ell : \ell \in L_1 \vee \ell \in L_2\})$
    b. *Concatenation:* $(L := \{\ell_1 \ell_2 : \ell_1 \in L_1 \wedge \ell_2 \in L_2\}$
    c. *Kleene Star:* L^* is empty or more words from L put together.

$$L^* = \bigcup_{n \geq 0} L^n, \text{ where } L^n = \begin{cases} \{\varepsilon\} & \text{if } n = 0 \\ LL^{n-1} & \text{otherwise} \end{cases}$$

4. *Rejection:* Nothing else is regular.

As an example, let $\Sigma = \{a, b\}$. The language $L = \{ab^n a : n \in \mathbb{N}\}$ is regular because:

- $\{a\}$ is regular (Rule 2, singleton is regular)
- $\{b\}$ is regular so $\{b\}^*$ is regular (Rule 3c, Kleene Star)
- Thus $\{a\} \cdot \{b\}^* \cdot \{a\}$ is regular (Rule 3b, concatenation)

We can express regular languages using regular expressions:

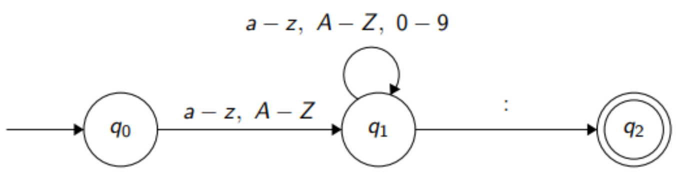
- Drop set notation for empty string and singletons: $\{\varepsilon\} \implies \varepsilon$, $\{a\} \implies a$.
- Union/Alternation with pipes: $L_1 \cup L_2 \implies L_1 \mid L_2$ or $L_1 + L_2$.
- Concatenation stays the same: $L_1 \cdot L_2$ or $L_1 L_2$.
- Empty language maintains the same notation of $\varnothing$.

For example, we can express the previous example as ab^*a.

3 Deterministic Finite Automata

A **DFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ: a finite non-empty set (alphabet).
- Q: a finite non-empty set of states.
- $q_0 \in Q$: a start state.
- $A \subseteq Q$: a finite set of accepting states.
- $\delta : (Q \times \Sigma) \to Q$: transition functions. [1]



- $\Sigma = \{$ ASCII characters$\}$
- $Q = \{q_0, q_1, q_2\}$
- q_0 is our start state
- $A = \{q_2\}$ (note: this is a set!)

- δ is defined by
  - $\delta(q_0, \text{letter}) = q_1$
  - $\delta(q_1, \text{letter or number}) = q_1$
  - $\delta(q_1, :) = q_2$
  - All other transitions go to an error state.

States can have labels inside the bubble; we refer to the states using these labels. For each character you see, follow the transition. If there is none, go to the error state (not shown in CS241). Once the input is exhausted, check if the final

To parse an entire string (rather than one character only), we can extend the definition of $\delta : (Q \times \Sigma) \to Q$ to a function defined over $(Q \times \Sigma^*)$ via:

$$\begin{aligned} \delta^* : (Q \times \Sigma^*) &\to Q \\ (q, \varepsilon) &\mapsto q \\ (q, aw) &\mapsto \delta^*(\delta(q, a), w) \end{aligned}$$

where $a \in \Sigma$ and $w \in \Sigma^*$ (aw is a concatenation). [2]

We say a DFA given by $M = (\Sigma, Q, q_0, A, \delta)$ **accepts a string** w iff $\delta^*(q_0, w) \in A$. The **language of a DFA** M is defined to be the set of all strings accepted by M

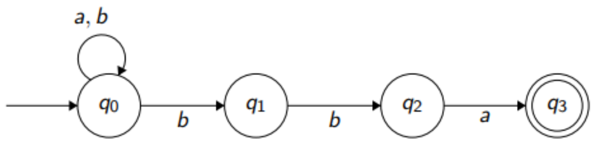
$$L(M) = \{w : M \text{ accepts } w\}.$$

A formal language L is regular if and only if $L = L(M)$ for some DFA M. That is, the regular languages are precisely the languages accepted by DFAs. [3]

4 Non-Deterministic Finite Automata

An **NFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ: a finite non-empty set (alphabet).
- Q: a finite non-empty set of *states*.
- $q_0 \in Q$: a *start* state.
- $A \subseteq Q$: a finite set of *accepting* states.
- $\delta : (Q \times \Sigma) \to 2^Q$: transition functions. [4]



Similarly, we can extend the definition of $\delta : (Q \times \Sigma) \to 2^Q$ to a function $\delta^* : (2^Q \times \Sigma) \to 2^Q$ via:

$$\begin{aligned} \delta^* : (2^Q \times \Sigma^*) &\to 2^Q \\ (S, \varepsilon) &\mapsto S \\ (S, aw) &\mapsto \delta^*\left(\bigcup_{q \in S} \delta(q, a), w\right) \end{aligned}$$

- $S \subseteq 2^Q$ is a set of states. Given the current state, we want to go to all next states.
- $\bigcup_{q \in S} \delta(q, a)$ denotes all paths, i.e., we want to go everywhere!

An NFA given by $M = (\Sigma, Q, a_0, A, \delta)$ **accepts a string** w if and only if $\delta^*(\{q_0\}, w) \cap A \neq \varnothing$. That is, there exists a path where starting at q_0 and consuming characters from w gets us to an accepting state.

For every NFA there is an equivalent DFA and vice-versa. Thus, the set of languages accepted by an NFA are precisely the regular languages.

To convert an NFA to a DFA:

1. Start with the state $S = \{q_0\}$.
2. From this state, go to the NFA and determine what happens on each $a \in \Sigma$ for each $q \in S$. The set of resulting states should become its own state in the DFA.
3. Repeat step 2 for each newly-created state until every possibility has been exhausted.
4. Accepting states are any states that include an accepting state of the original NFA.

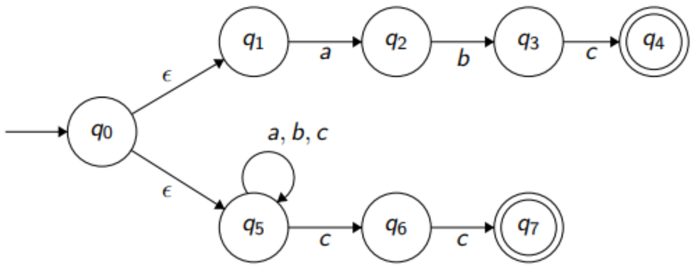
5 NFA with Epsilon-Transitions

To gain new computing powers from an NFA, we permit state changes without reading a character; we call them **ε-transitions.**

An **ε-NFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ: a finite non-empty set (alphabet) that **does not contain** ε.
- Q: a finite non-empty set of *states*.
- $q_0 \in Q$: a *start* state.
- $A \subseteq Q$: a finite set of *accepting* states.
- $\delta : (Q \times \Sigma \cup \{\varepsilon\}) \to 2^Q$: transition functions.

We accepting ε as these ε-transistions make it trivial to take the union of two NFAs. For example, we can use the following ε-NFA to solve the problem: $L = \{abc\} \cup \{w : w \text{ ends with } cc\}$.



Every ε-NFA has a corresponding DFA. Thus, every language recognized by an ε-NFA is regular.

We can use *structural induction* to show that an ε-NFA exists for every regular expression (by showing there exists an ε-NFA which parses $\varnothing$, $\{\varepsilon\}$, $\{a\}$, $L_1 \cup L_2$, $L_1 L_2$ and L^* correctly).

6 Maximal Munch and Simplified Maximal Munch

Given a regular language L, we want to determine if a given word w is in LL^* (or in other words, is $w \in L^* \setminus \{\varepsilon\}$?) Note that we dropped ε as we don't consider the empty program as being valid. The general idea is, *we consume the largest possible token that makes sense; produce the token and then proceed.*

Maximal Munch: we consume characters until there is no longer a valid transition. If there are characters left to consume, backtrack to the last valid accepting state and resume.

Algorithm 1 Maximal Munch

```
s = $q_0$, $t_a = \epsilon$, $t_{cur} = \epsilon$
str = input, $pos = 0$, $posAccepting = -1$
while $pos != len(str)$ do
    c = str[pos], $s = \delta(s, c)$, $t_{cur} = t_{cur} + c$
    if $s ==$ ERROR then
        if $t_a = \epsilon$ then
            Fatal Error
        end if
        Output $t_a$
        $s = q_0$, $t_a = \epsilon$, $t_{cur} = \epsilon$, pos = posAccepting
    else if $s \in A$ then
        posAccepting = pos
        $t_a = t_{cur}$
    end if
    pos = pos + 1
end while
if $s \in A$ then
    Output $t$ and Accept
else
    Reject/Crash/Fatal Error
end if
```

Simplified Maximal Munch: we consume characters until there is no longer a valid transition. If we are currently in an accepting state, produce the token and proceed, otherwise go to an error state.

Algorithm 2 Simplified Maximal Munch

```
$s = q_0$
$t_a = \epsilon$
while not EOF do
    c = read_char()
    if $\delta(s, c) ==$ ERROR then
        if $s \in A$ then
            Output $t_a$
            $s = q_0$, $t_a = \epsilon$.
        else
            Reject/Crash/Fatal Error
        end if
    else
        $s = \delta(s, c)$
        $t_a = t_a + c$
    end if
end while
if $s \in A$ then
    Output $t_a$ and Accept
else
    Reject/Crash/Fatal Error
end if
```

7 Context Free Grammar

A **CFG** is a 4-tuple (N, Σ, P, S) where

- N is a finite non-empty set of non-terminal symbols,
- Σ is an alphabet; a set of non-empty terminal symbols,
- P is a finite set of productions, each of the form $A \to \beta$ where $A \in N$ and $\beta \in (N \cup \Sigma)^*$,
- $S \in N$ is a starting symbol.

We have the following conventions:

- *Terminals:* lower case letters from the start of the alphabet, say $a, b, c, \ldots$, are elements of Σ.
- *Words:* lower case letters from the end of the alphabet, say $x, y, z, \ldots$, are elements of Σ^*.
- *Non-terminals*: upper case letters from the start of the alphabet, say $A, B, C, \ldots$, are elements of N.
- S is always our starting symbol.
- We set $V = N \cup \Sigma$ to denote the *vocabulary*, that is, the set of all symbols in our language; Greek letters, $\alpha, \beta, \gamma, \ldots$, are elements of $V^* = (N \cup \Sigma)^*$.

Over a CFG (N, Σ, P, S), we say that

- $\alpha \Rightarrow \beta$ (α derives β) if and only if β can be obtained from α using a rule from P.
- $\alpha A \beta \Rightarrow \alpha \gamma \beta$ if and only if there is a rule $A \to \gamma$ in P.
- $\alpha \Rightarrow^* \beta$ if and only if a *derivation* exists, that is, there exists $\delta_i \in V^*$ for $0 \leq i \leq k$ such that $\alpha = \delta_0 \Rightarrow \delta_1 \Rightarrow \cdots \Rightarrow \delta_k = \beta$. Note that, k can be 0.

We define the **language of a CFG** to be $L(G) = \{w \in \Sigma^* : S \Rightarrow^* w\}$. A language is **context-free** if and only if there exits a CFG G such that $L = L(G)$.

Every regular language is context free:

- $\varnothing : (\{S\}, \{a\}, \varnothing, S)$.
- $\{\varepsilon\} : (\{S\}, \{a\}, S \to \varepsilon, S)$.
- $\{a\} : (\{S\}, \{a\}, S \to a, S)$.
- $\{a\} \cup \{b\} : (\{S\}, \{a, b\}, S \to a|b, S)$.
- $\{ab\} : (\{S\}, \{a, b\}, S \to ab, S)$.
- $\{a\}^* : (\{S\}, \{a\}, S \to Sa|\varepsilon, S)$.

8 Ambiguous/Unambiguous Grammars

A grammar for which some word has more than one distinct leftmost derivation/rightmost derivation/parse tree is called *ambiguous*. We can solve the problem by forcing the derivation to be left/right associative.

- **Ambiguous**
  - $S \to a \mid b \mid c \mid SRS$
  - $R \to + \mid - \mid * \mid /$
- **Forcing Right Associative**
  - $S \to LRS \mid L$
  - $L \to a \mid b \mid c$
  - $R \to + \mid - \mid * \mid /$
- **Forcing Left Associative**
  - $S \to SRL \mid L$
  - $L \to a \mid b \mid c$
  - $R \to + \mid - \mid * \mid /$

We can use the above to create a grammar for BEDMAS rules:

$$\begin{aligned}
&S \to SPT \mid T \\
&T \to TRF \mid F \\
&F \to a \mid b \mid c \mid (S) \\
&P \to + \mid - \\
&R \to * \mid /
\end{aligned}$$

If L is a context-free language, there does NOT always exists an umambiguous grammar such that $L(G) = L$.

We cannot determine whether or not a grammar is ambiguous using a computer program.

9 Parsing

Given a CFG $G = (N, \Sigma, P, S)$ and a terminal string $w \in \Sigma^*$, we want to find a **derivation**, that is, the steps such that $S \implies \cdots \implies w$ or prove that $w \notin L(G)$ using **pushdown automaton**, DFAs with an additional stack that we can process in LIFO order.

We have two strategies:

1. Forward/Top-down: start with S and try to get to w, e.g., $LL(1)$.
2. Backward/Bottom-up: start with w and figure out how we could've gotten to w, e.g., $LR(0)$ and $SLR(1)$.

10 Top-Down Parsing

Start with S and store immediate derivations in a stack and then match characters to w.

We augment our grammar to include $\vdash$ and $\dashv$ symbolizing the beginning and end of the file, respectively. We also include a new start state S' to begin our parsing:

$$G = (N, \Sigma, P, S) \implies G = (N \cup \{S'\}, \Sigma \cup \{\vdash, \dashv\}, P \cup \{S' \to \vdash S \dashv\}, S')$$

A grammar is called **LL(1)** (scan left to right, leftmost derivation, 1-step lookahead) if and only if each cell of the predictor table contains at most one entry.

10.1 Lookahead Table

To help us determine which path to take, we define the following:

$$\begin{aligned}
\mathit{Predict}(A, a) &= \{A \to \beta \mid a \in \mathit{First}(\beta)\} \cup \{A \to \beta : \beta \text{ is nullable and } a \in \mathit{Follow}(A)\} \\
\mathit{First}(\beta) &= \{a \in \Sigma' \mid \exists \gamma \in V^* : \beta \Rightarrow^* a\gamma\} \\
\mathit{Nullable}(\beta) &: \text{Boolean function, True} \iff \beta \Rightarrow^* \epsilon \\
\mathit{Follow}(A) &= \{b \in \Sigma' \mid \exists \alpha, \beta \in V^* S' \Rightarrow^* aAb\beta\}
\end{aligned}$$

$\mathit{Predict}(A, a)$: production rules that apply when $A \in N'$ is on the stack, $a \in \Sigma'$ is the next input character.

$\mathit{First}(\beta) \subseteq \Sigma'$: set of characters that can be the first letter of a derivation starting from $\beta \in V^*$.

$\mathit{Nullable}(\beta) \subset \{\mathit{True}, \mathit{False}\}$: $\beta \in V^*$ is **nullable** iff $\beta \Rightarrow^* \epsilon$.

$\mathit{Follow}(A)$: for any $A \in N'$, this is the set of elements of Σ' that can come immediately after A in a derivation starting from S'.

10.2 Cheatsheet

	Rules
Nullable	$\mathit{Nullable}(\varepsilon) = \mathrm{T}, A \to \varepsilon \implies \mathit{Nullable}(A) = \mathrm{T}$ $(A \to B_1 \cdots B_n) \wedge (\forall i : \mathit{Nullable}(B_i) = \mathit{True}) \implies \mathit{Nullable}(A) = \mathrm{T}$
First	$A \to a\alpha \implies a \in \mathit{First}(A)$. $A \to B_1 \cdots B_n \Rightarrow \mathit{First}(A) = \mathit{First}(A) \cup \mathit{First}(B_i) \forall i \in \{1, \ldots, n\}$ until $\mathit{Nullable}(B_i) = \mathrm{F}$
Follow	$A \to \alpha B \beta \implies \mathit{Follow}(B) = \mathit{First}(\beta)$ $(A \to \alpha B \beta) \wedge (\mathit{Nullable}(\beta) = \mathit{True}) \implies \mathit{Follow}(B) = \mathit{Follow}(B) \cup \mathit{Follow}(A)$
Predict	$\mathit{Predict}(A, a) = \{A \to \beta : a \in \mathit{First}(\beta)\} \cup \{A \to \beta : \mathit{Nullable}(\beta) = \mathit{True} \wedge a \in \mathit{Follow}(A)\}$

1. *Given a state and a symbol of our alphabet, which state should we go to?* ↩

2. *Basically, when processing a string, process a letter first then process the rest of a string.* ↩

3. *This can be seen as the alternative definition to regular languages.* ↩

4. *Given a state and a symbol of our alphabet, what state(s) should we go next? Note that* 2^Q *denotes the poewr set of* Q*, which allows us to go to multiple states at once.* ↩