Regular Languages and Finite State Machines

Remington Greko, Tyler Gutowski, Spencer Hirsch, Thomas Johnson

February 6, 2023

1. **Read the Wikipedia article on regular grammars. Summarize the saliet points.**

   Briefly a regular grammar extends from a formal grammar. A formal grammar is given by the definition of:

   $$G = (V, T, S, P)$$

   with

   V: A finite set of **variables**.

   T: A finite set of **symbols**.

   S: $S \in V$ being a **special** symbol called the **start symbol**.

   P: Being a set of **productions**.

   With productions showing how symbols from V, and S may be used to create *strings*.

   $$G = (\{S, A\}, \{a, b\}, S, P)$$

   With productions being:

   $S \Rightarrow ab$

   $S \Rightarrow aS$

   $S \Rightarrow bS$

   $S \Rightarrow \lambda$

   Knowing this we can look at **regular grammars**

- **Strictly right-regular grammars**

  These follow the production rule of $S \Rightarrow bA$ where non terminal symbols from V go on the right hand side

- **Strictly left-regular grammars**

  These follow the production rule of $S \Rightarrow Ab$ where non terminal symbols from V go on the left hand side

- **Extended regular grammars**

  These grammars will follow the production rules of either right, or left, $S \Rightarrow Ab$, where $b \in \sum^*$.

2. **What is a Deterministic Finite Accepter (DFA)?**

A Deterministic Finite Accepter is a machine that can process an input string from left to right. A Finite Accepter is deterministic when there is only one thing that it can do for an input symbol. The DFA will either accept or reject the string. Once a String is accepted once, it will always be accepted.

A DFA can only read left to right, just as traditional in the English language. The DFA can only see one specific element of a string at a time, it cannot go backwards, nor skip ahead. A DFA also has a specific number of internal states, each different based on its current situation, such as when beginning a string.

The example from the book is a great example of a Deterministic Finite Accepter,

$$M = (Q, \Sigma, \delta, q_0, F)$$

where

Q is a finite set of **internal states**,

Σ is a finite set of symbols called the **input alphabet**,

δ: $Q \times \Sigma \rightarrow Q$ is a total function called the **transition function**,

$q_0 \in Q$ is the **initial state**,

$F \subseteq Q$ is a set of **final states**.

3. **What is a Non-Deterministic Finite Accepter (NFA)?**

A non-deterministic finite acceptor can also be defined with the same formal definition of a deterministic finite accepter.

$$M = (Q, \Sigma, \delta, q_0, F)$$

However an NFA has three main differences from a DFA.

- The range of δ is a subset of Q. Meaning the range is 2^{δ}
- λ is allowed as input for the transition function $\delta(q_1, \lambda)$
- The transfer function may equal an empty set. $\delta(q_i, a) = \emptyset$

A brief encapsulation of the above topics is that an NFA has the power of choice. Given an input character the transition function has multiple internal states to choose from. A NFA has clear advantages over a DFA for its more practical applications. Most complex problems can't always be solved deterministically without performing exhaustive backtracking based searches due to their linear design. This makes a NFA ideal for simulation of search-backtracking

4. **Explain why the languages accepted by DFAs and NFAs are the equivalent.**

Any language accepted by a DFA can also be accepted by an NFA, and vice versa. We can prove this through the use of the "subset construction algorithm." The core principle of this algorithm is the DFA simulates the NFA by keeping track of the every possible state. Each state of the DFA corresponds to a subset of the sets of the NFA.

As long as the languages are equivalent then both a DFA and an NFA can be used when using the language. Although a DFA can only consist of a fixed number of processes, those processes are a subset of the NFA processes, therefore languages accepted by both the DFA and the NFA are equivalent.

5. **Give a recursive definition of *regular expression* over an alaphabet Σ.**

A regular expression can be constucted by repeatedly applying recursive rules to primitive constituents. The algorithmic process is defined below.

Let Σ be a given alphabet. Apply the following steps:

(a) $\emptyset$, λ, and $\alpha \in \Sigma$ are all regular expressions. All of these are **primitive regular expressions**.

(b) If r_1 and r_2 are regular expressions, so are $r_1 + r_2$, $r_1 \cdot r_2$, r_1^*, and (r_1).

(c) A string is a regular expression iff it can be derived from the primitive regular expressions by a finite number of applications of the rules in step (b).

6. **Confirm you know how to use operating system commands to find regular expressions in a file.**

```
C:\Users\tygut\Documents\formal_language(main -> origin)
λ grep -E "{[[:digit:]]}" regular_languages.tex
        \item If \textit{r}\textsubscript{1} and \textit{r}\textsubscri
        regular expressions, so are \textit{r}\textsubscript{1} +
        \textit{r}\textsubscript{2}, $\textit{r}\textsubscript{1} \cdot
        \textit{r}\textsubscript{2}$, $r_1^*$, and (\textit{r}\textsubs
```

The operating system command used, **grep -E "{[[:digit:]]}" regular_languages.tex** uses **grep** (global regular expression print) with the parameter **-E**, which is shorthand for –**regex-extended**. The regex phrase **"{[[:digit:]]}"** is then searched, which is any numeral inside of curly braces. As shown on the four lines below the command, there were several instances of a numeral within curly braces, all of which were a part of **textsubscript**{}.

Name	**Section**
Remington Greko	Question 5
Tyler Gutowski	Question 6
Spencer Hirsch	Question 2 and 4
Thomas Johnson	Question 1 and 3