

# Chapter 2

Daniel Lee

May 7th, 2022

## Introduction

- A compiler scans an input of characters and outputs a stream of words labelled by syntactic category
- A microsyntax is used to group words that have meaning within the source language
- Some words such as keywords have special meaning, which makes them reserved
- An example of this would be the *while* and *static* keywords in the Java programming language
- To recognize keywords, the scanner can either use dictionary lookup or encode keywords directly into microsyntax
- The simple lexical structure of programming languages lends itself to efficient scanners

## Recognizing Words

- When we are parsing words we can view the parsing process as a series of if-else statements or a state machine
- Transition diagrams often provide a simple means of formalizing the abstractions a compiler may need to implement them
- $S$  is the finite set of states in the recognizer, alongside with error state  $s_e$
- $\Sigma$  is the finite alphabet recognized by the recognizer
- $\delta(s, c)$  is the transition function, it maps the value of state  $s$  and  $c$ , into some state
- In state  $s_i$  with transition character  $c$ , the state makes the following transition  $s_i \rightarrow_c \delta(s_i, c)$
- $s_0 \in S$  refers to initial state
- $S_a (S_a \subseteq S)$ , is the set of accepting states

### Example:

$S = \{s_0, s_1, s_2, s_3, \dots, s_{10}, s_e\}$   
 $\Sigma = \{e, h, i, l, n, o, t, w\}$   
 $\delta =$   
 $\{s_0 \rightarrow_n s_1, s_0 \rightarrow_w s_6, s_1 \rightarrow_e s_2, s_1 \rightarrow_o s_4, s_2 \rightarrow_w s_3$   
 $s_4 \rightarrow_t s_5, s_6 \rightarrow_h s_7, s_7 \rightarrow_i s_8, s_8 \rightarrow_l s_9, s_9 \rightarrow_e s_{10}$   
 $s_0 = s_0$   
 $S_A = \{s_3, s_5, s_{10}\}$

### More complex words:

- For more complex words we can have the state machine accept multiple inputs
- We can vastly simplify state machines by using cycles

### Practice Problems:

- Problem 1: A six-character identifier consisting of alphanumeric characters followed by zero to five-alpha numeric characters
  - $S = \{s_0, s_1, s_e\}$
  - $\Sigma = a = \text{set of all-alphabet}, b = \text{set of all alphanumeric}$
  - $s_0 = s_0$
  - $\delta = \{s_0 \xrightarrow{a} s_1, s_1 \xrightarrow{b} s_1\}$
  - $S_A = s_1$
- Problem 2:
  - $S = \{s_0, s_1, s_2 s_e\}$
  - $\Sigma = (, )$
  - $s_0 = s_0$
  - $S_A = \{s_2\}$
  - $\delta = \{s_0 \xrightarrow{(} s_1, s_1 \xrightarrow{)} s_2, s_2 \xrightarrow{(} s_1\}$
- Problem 3: A Pascal comment which consists of  $\{$ , zero or more characters from the alphabet, and closed by  $\}$ :
  - $S = \{s_0, s_1, s_2\}$
  - $\Sigma = \{\}, \{a...z, A...Z, 0...9\}$
  - $s_0 = s_0$
  - $S_A = \{s_3\}$
  - $\delta = \{s_0 \xrightarrow{\{ } s_1$   
 $s_1 \xrightarrow{(a...z, A...Z, 0...9)} s_1$   
 $s_1 \xrightarrow{\{ } s_2$

### Regular Expression

- The set of all words accepted by a finite automaton,  $F$ , forms a language  $L(F)$
- For any FA, we can describe describe the language using regular expression or RE
- The language consists of single word "new" can be described as RE, new

- A language consisting of two words, new or while can be represented as RE  $\text{new|while}$
- new or not can be represent by RE,  $\text{n(ew|ot)}$
- Let us consider the example of punctuation marks, a REs for punctuation may appear such as:  $\text{: ; ? = > ( ) []}$
- Keywords may have an expression such as this: if while this integer instance of
- more complex RE:  $0|(0|1|2|3|4|5|6|7|8|9)(0|1|2|3|4|5|6|7|8|9)^*$
- **The following operator is called a kleen operator and indicates there can be zero or more instances of a RE**

### Formalizing notes for regular expressions:

- Given a regular expression  $r$ , we can denote the Language it describes as  $L(r)$
- An RE is made up of 3 operations:
- Alternation: The alternation or union of two sets  $R$  and  $S$  denoted  $R|S$  or  $\{x|x \in R \text{ or } x \in S\}$
- Concatenation: The concatenation of two sets  $RS$  contains all strings formed by prepending an element of  $R$  onto one from  $S$ , or  $\{xy|x \in R \text{ and } y \in S\}$
- Closure: The kleene closure of a set  $R$ , denoted by  $R^*$  is  $\cup_{i=0}^{\infty} R^i$  is a concatenation of  $R$  with itself zero or more times
- Sometimes we can use notation for finite closure if a set is concatenated multiple times:  
 $(R|RR|RRR)$
- Positive Closure is Denoted if  $RR^*$ 
  - If  $a \in \Sigma$ , then  $a$  is also an RE denoting set containing only  $a$
  - If  $r$  and  $s$  are RES, denoting sets  $L(r)$  and  $L(s)$  then  $r | s$  is a RE denoting the union, or alternation of  $L(r)$  and  $L(s)$   
Similarly  $rs$  is an RE denoting the concatenation of  $L(r)$  and  $L(s)$   
 $r^*$  is an RE denoting Kleene closure of  $L(r)$
  - $\epsilon$  represents a RE of an empty string
  - Parentheses have the highest precedence, followed by closure, concatenation, and alternation

**Example:**

- Imagine a language in keywords start with a letter in the English alphabet and can be then followed by a sequence of alphanumeric character. We can represent the following keyword using RE:  

$$([A...Z]|([a...z]))([A...Z]|([a...z])|([0...9]))^*$$
- Let's consider another example one in which, we are representing unsigned integers:  

$$(0|([1...9])([0...9]))^*$$
- Unsigned real number:  

$$(0|([1...9])([1...9])^*)(\epsilon|. [0...9]^*)$$
- Quoted strings using complement:  
 A quoted String in a programming language is often composed of a " followed by ", in between these two characters any characters can appear. In theory we could write a regular expression that contains all the possible characters but this is impractical. To circumvent this issue we can use the complement operator:  

$$"(\wedge)"$$
- Comments can often appear in many forms:  

$$(/\wedge \backslash n)^* | / * (\wedge * | \wedge * /)^* * /$$

**Closure properties of RE:**

- Many regular expressions are closed under many operations, i.e. if we apply an operation to a RE we get a RE
- Some obvious examples are concatenation, union, and closure
- Imagine we have a collection of regular expressions to describe syntactic categories in a language:  $a_0, a_1, \dots, a_n$
- To describe all the valid words in a language we can use the RE:  $a_0|a_1|a_2|\dots|a_n$
- Closure under union suggests that any finite language is a regular language and can be arranged in alternation
- Closure under concatenation also allows us to build complex REs from simpler one's by concatenating them
- REs are closed under complements

**Practice Problems:**

- Chapter 2: pg 42
- Problem 1:

- $a_0 = [A...Z], a_1 = [a...z], a_2 = [0...9]$   
 $(a_0|a_1)(a_0a_1a_2)^5$
- Problem 2:
- $a_0 = ^\wedge, a_1 = ^\text{''}$   
 $a_1(a_0|a_1a_1)^*a_1$

## From Regular Expression to Scanner:

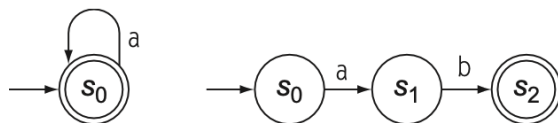
- Up to this point we have covered a decent amount of notation, so what is the point of all this notation
- In this section we will be learning about deterministic FA(DFA) and non-deterministic FA(NFA)

## Nondeterministic Finite Automata

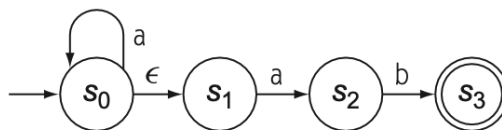
- Recall that an empty string  $\epsilon$  is a RE, but many FAs did not include the empty string within the state diagrams
- So what role does  $\epsilon$  play in FA?
- The answer lies in using  $\epsilon$  to build more complex state machines
- For instance, we can use  $\epsilon$  to represent concatenation of mn using  $\epsilon$

$$\rightarrow s_0 \rightarrow_m s_1 \rightarrow_\epsilon s_2 \rightarrow_n s_3$$

- $\epsilon$  can often complicate how our model work
  - Consider the following two FAs



- Utilizing the  $\epsilon$  transition we can add complexity to our model to form  $a^*ab$



- The following machine is called a NFA, due to the fact that for a specific character there can be transitions for same state depending on what character follows the input character
- A DFA on the contrary can only have a single transition for a single character for a state

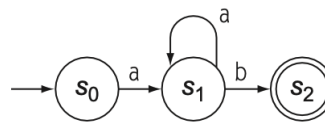
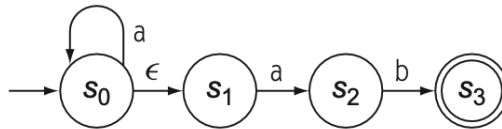
#### A Set of Models for NFA:

- Each time a NFA makes a nondeterministic choice, it follows the transition that leads to accepting state of the input string
- Each time the NFA makes non-deterministic choice, it replicates itself to pursue each possible translation
- In this state, each possible translation is pursued and the current active state is called the *configuration*

#### Equivalence of NFAs:

- NFAs and DFAs are equivalent in their expressive power
- Any DFAs is a special case of an NFA
- Conversely any NFA can be emulated by a DFA
- Let us consider the state of a NFA when it has reached some point in input string
- If there are  $n$  states and  $|\Sigma|$  character then by rule of product there are  $|\Sigma|^n$  configurations
- To simulate the behavior of the NFA, we need a DFA for each configuration of NFA thus have more exponentially more states than NFA
- This can make a DFA expensive from space standpoint

- Examine the two figures below:



- Notice how both expressions are equivalent but instead of there of being two transition state for same characters one simply moves on to the next state and have self-loop

## Regular Expression to NFA: Thompson's Construction

- Thompson's construction can be used to build NFA from RE
- In a sense they are a template for building FAs and can be viewed as a combination of concatenation, alternation, and closure
-