

1 Formal Languages

In the last module we were able to discuss the **specification** and **recognition** of MIPS programs without making things formal. We took the MIPS reference sheet as the specification for what constitutes valid MIPS instructions and discussed the rules that the assembler must implement to recognize whether input to the assembler is a valid MIPS program. As we move to higher-level languages, with more complex syntax and rules, it gets harder to specify the language and implement a recognition algorithm. In this module, we begin by learning the formal theory of string recognition and the general principles that work in the context of programming languages. We begin with a few definitions:

Definition 1 An **alphabet** is a non-empty finite set of symbols often denoted by Σ (capital sigma).

Examples:

- $\Sigma = \{a, b, c, \dots, z\}$ lowercase English alphabet.
- $\Sigma = \{0, 1\}$ alphabet of binary digits.
- $\Sigma = \{0, 1, 2, \dots, 9\}$ alphabet of decimal digits.
- $\Sigma = \{\text{WHILE, IF, ELSE, VAR, NUM, EQUALS, BECOMES, PLUS, ...}\}$ alphabet of elements in a programming language.

Notice the last example above; alphabet “symbols” are usually a single character or letter, but in some contexts, they might consist of multiple characters. For example, “WHILE”, in the last example above, is a single symbol and not five symbols. In these contexts, the formal entity we call an “alphabet” is actually more like a lexicon (i.e., a dictionary).

Definition 2 A **word** (or **string**) w is a finite sequence of symbols chosen from Σ . The set of all words over an alphabet Σ is denoted by Σ^* .

- The **length of a word** w is denoted by $|w|$.
- ϵ (epsilon) is the empty word; it is in Σ^* for any Σ and $|\epsilon| = 0$. Note that ϵ is just used as a placeholder, as the actual form of the empty word is empty. The empty word does not actually contain an epsilon symbol.

Examples:

- For $\Sigma = \{0, 1\}$, words include $w = 011101$ or $x = 1111$. Note $|w| = 6$ and $|x| = 4$.
- For $\Sigma = \{a, b, c, \dots, z\}$, words include `hello` and `lbyjx`.
- For $\Sigma = \{\text{WHILE, IF, ELSE, VAR, NUM, EQUALS, BECOMES, PLUS, ...}\}$, an example of a word over this alphabet is `WHILE VAR EQUALS NUM VAR BECOMES VAR PLUS NUM`, representing a simple while loop in this programming language. The length of this word is 9.

Note: For our course, assume Σ will never contain the symbol ϵ , so ϵ always represents the empty string.

Definition 3 A **language** is a set of strings.

Examples:

- $L = \emptyset$ or $\{\}$, the empty language.
- $L = \{\epsilon\}$ the language consisting of the empty string.
- $L = \{ab^na : n \in \mathbb{N}\}$ 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 .
- $\{., -\}$, $L = \{ \text{words in Morse Code} \}$
- $L = \{ \text{syntactically valid programs} \}$ over $\Sigma = \{\text{WHILE, IF, ELSE, VAR, NUM, EQUALS, BECOMES, PLUS, ...}\}$. For example, this language would not include WHILE PLUS PLUS.

A language may be finite or infinite—that is, the set that defines L may be finite or infinite—though finite languages are rarely interesting. A language L over alphabet Σ is by definition a subset of Σ^* , and Σ^* is itself a language.

1.1 Recognition

Definition 4 A **recognition algorithm** is a decision algorithm that takes the specification of a language and an input word and answers whether the word satisfies the specification, i.e., whether the word is in the language.

While we define recognition as the task of determining whether a word is in the language, our ultimate goal will be to determine whether an input program satisfies the specification of the programming language. We also make the distinction that a recognition algorithm is not a compiler for a language. Part of what a compiler does is recognition, but then it also **translates** the program into a different output language (e.g., for the assembler, the assembler generated MIPS machine instructions).

The obvious question is, given a language, how hard is it to determine if an input word belongs to it? In other words, how hard is it to write a recognition algorithm for a language? The answer: it depends on the language.

- $L = \{ab^na : n \in \mathbb{N}\}$ Easy
- $L = \{ \text{valid MIPS assembly programs} \}$ Medium
- $L = \{ \text{valid Java/C/C++ Programs} \}$ Hard
- $L = \{ \text{set of programs that return "Yes" as a decision problem} \}$ Impossible!

We can begin to classify languages based on how hard it is to recognize them. In order of relative difficulty: finite, regular, context-free, context-sensitive, recursive/decidable and undecidable. We will formally discuss Finite and Regular languages in this module and discuss Context-Free languages in the next module. We touch upon Context-Sensitive languages, but never treat them formally in CS241.

2 Finite Languages

Since a finite language is a finite set of words, one way to specify the language is to list all the words (although more efficient ways might exist). A recognition algorithm can compare the input word with each. While inefficient for large languages, it is still theoretically possible. Sometimes, there are more efficient ways.

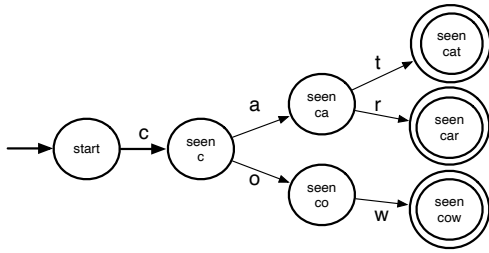
Example: Suppose we have the language, $L = \{\text{cat}, \text{car}, \text{cow}\}$. Write a program that determines whether or not $w \in L$ given that each character of w is scanned exactly once without the ability to store previously seen characters.

We describe below the algorithm for the program. Notice how it only inspects each character in the input once and makes a decision based on the character and the current state of the program.

Algorithm 1 Algorithm to recognize L

```
1: Scan input left to right
2: if first char is c then
3:   if next char is a then
4:     if next char is t then
5:       if no next char then
6:         Accept
7:       else
8:         Reject
9:       end if
10:    else if next char is r then
11:      if no next char then
12:        Accept
13:      else
14:        Reject
15:      end if
16:    else
17:      Reject
18:    end if
19:  else if next char is o then
20:    if next char is w then
21:      if no next char then
22:        Accept
23:      else
24:        Reject
25:      end if
26:    else
27:      Reject
28:    end if
29:  else
30:    Reject
31:  end if
32: else
33:   Reject
34: end if
```

Reading through the algorithm to convince yourself that it recognizes only words we intended it to recognize likely took time. We can represent such algorithms more concisely using **state diagrams**.



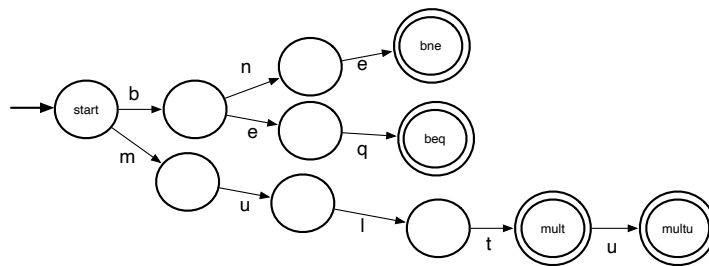
- Each bubble represents a **state**: configuration of the program based on the input seen/scanned so far.
- There is a unique **start** state. In this course we identify the start state with an arrow from nowhere coming into it, but other conventions exist in other contexts. In the example to the left, the start state is labeled “start”, but that is not necessary.
- Arrows from state to state are labelled and represent **transitions** based on the next input symbol.
- At any state, if a transition is not defined on the next input symbol, the state transitions to an implicit (not shown) **error** state.
- Some states are **accepting** states (shown with two circles).

We can use the state diagram to implement our recognition algorithm. Starting at the start state, we transition states based on the characters in the input word. If we get stuck (no transition on the next input character), we reject the word. If the input is exhausted, we accept the word if we are at an accepting state. Otherwise, we reject.

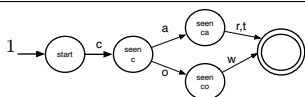
Can you think of a state diagram for the language $L = \{\text{cat}, \text{car}, \text{cow}\}$ that has fewer states than the above example? Hint: a recognition algorithm needs to only answer whether the input word satisfies the specification not what the accepted word was.

Example: Suppose we have the language, $L = \{\text{bne}, \text{beq}, \text{mult}, \text{multu}\}$. Write a program that determines whether or not $w \in L$ given that each character of w is scanned exactly once without the ability to store previously seen characters.

We use the following state diagram to describe a program to recognize words in this language.



Notice that this state diagram recognizes four of the eighteen instruction keywords in MIPS Assembly Language.



We have simplified the state diagram by combining the three accepting states into one. Although the state diagram no longer has a way to distinguish which word was accepted, it still recognizes the same language.

Programming languages do not admit only finitely many words (e.g., we did not mention assembly language labels in the above). Despite the simplicity of these examples of finite languages, state diagrams can easily be generalized to recognize a larger class of languages known as *regular languages*.

3 Regular Languages

Definition 5 A language over an alphabet Σ is a **regular language** if:

1. It is the empty language, $\{\}$, or the language consisting of the empty word, $\{\epsilon\}$.
2. It is a language of the form $\{a\}$ for some $a \in \Sigma$.
3. It is a language built using the union, concatenation or Kleene star (pronounced klay-nee) of two regular languages.

3.1 Union, Concatenation, Kleene Star

Let L , L_1 and L_2 be regular languages. Then the following are regular languages:

- Union: $L_1 \cup L_2 = \{x : x \in L_1 \text{ or } x \in L_2\}$
Example: $L_1 = \{up, down\}$, $L_2 = \{hill, load\}$ then $L_1 \cup L_2 = \{up, down, hill, load\}$.
- Concatenation: $L_1 \cdot L_2 = L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$
Example: $L_1 = \{up, down\}$, $L_2 = \{hill, load\}$ then $L_1 \cdot L_2 = \{uphill, upload, downhill, download\}$
Similar to multiplication, the dot is often left implicit, i.e., we write $L_1 L_2$ instead of $L_1 \cdot L_2$.
- Kleene star $L^* = \{\epsilon\} \cup \{xy : x \in L^*, y \in L\} = \bigcup_{n=0}^{\infty} L^n$ where

$$L^n = \begin{cases} \{\epsilon\}, & \text{if } n = 0; \\ LL^{n-1}, & \text{otherwise.} \end{cases}$$

Equivalently, L^* is the set of all strings consisting of 0 or more occurrences of strings from L concatenated together. Written informally, $L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \dots$

Example: $L = \{a, b\}$ then $L^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, \dots\}$

What does $L \cdot \{\}$ equal?

2

Let $\Sigma = \{a, b\}$. Explain why the language $L = \{ab^n a : n \in \mathbb{N}\}$ is regular.

3

²It is the empty language, $\{\}$. Recall that the concatenation of L_1 and L_2 , i.e., $L_1 \cdot L_2$, has words where the first part of each word comes from L_1 and the second part must come from L_2 . In this example, L_2 is the empty language, and the concatenation of any language to the empty language will produce an empty language.

³ $\{a\}$ is regular. $\{b\}^*$ is also regular since $\{b\}$ is regular and the Kleene star of a regular language is a regular language, i.e., regular languages are closed under Kleene star. Therefore, the concatenation $\{a\} \cdot \{b\}^* \cdot \{a\}$ must also be regular.

3.2 Regular Expressions

Those of you who took CS246, got a practical introduction to writing regular expressions while using the `egrep` tool. The notation for writing regular expressions is very similar except we drop the set notation. As examples:

- $\{\epsilon\}$ becomes ϵ (and similarly for other singletons).
- $L_1 \cup L_2$ becomes $L_1 \mid L_2$.
- Concatenation is still \cdot or implicit.
- The empty language maintains the same notation of \emptyset .

The order of operations is $*$, \cdot then \mid . (Kleene star, concatenation then union). For example, the language $L = \{ab^na : n \in \mathbb{N}\}$ is written as ab^*a as a regular expression.

What does the expression $aa \mid b^*$ match? Does it match abb ? Does it match the empty word?

4

Example: The regular expression $b^*ab^*(ab^*ab^*)^*$ represents a language over $\{a,b\}$ where words in the language have an odd number of `as`.

Write a regular expression for a language over $\{a,b\}$ where words in the language have an even number of `as`.

5

Note that our formal definition of regular expressions **does not include** Extended Regular Expressions or Perl Regular Expressions. Many of these extensions are merely there as a convenience. For example, extended regular expressions support the $+$ operator, which is 1 or more concatenations of the previous subexpression, as opposed to Kleene star which is 0 or more concatenations. In formal regular expressions, $+$ can be represented using concatenation and Kleene star (e.g., a^+ is the same as aa^*).

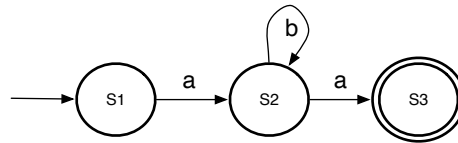
Although most extensions to regular expressions can be simplified in this way, some forms of “regular” expressions aren’t actually regular, because they have extensions that cannot be simplified in this way. By definition, all regular languages can be expressed as formal regular expressions.

⁴Due to order of operations, this is equivalent to $(aa) \mid (b^*)$. So, the expression only matches the word `aa`, or words in b^* (`ϵ` , `b`, `bb`, `bbb` ...). In particular it does not match `abb` but does match the empty word.

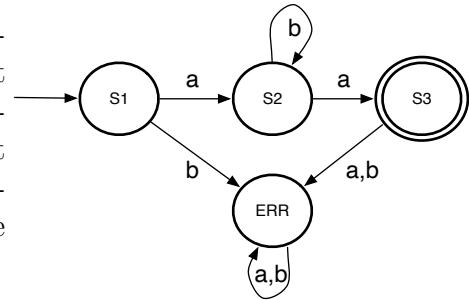
⁵The regular expression $b^*(ab^*ab^*)^*$ defines the language where words in the language have an even number of `as`. The regular expression $b^*(ab^*a)^*b^*$ however, does **not** capture all possible words with even number of `as`, e.g., `aabaa` has an even number of `as` but is not in the language defined by this regular expression.

3.3 Deterministic Finite Automata

We can use state diagrams similar to the ones discussed for finite languages to pictorially represent regular languages. The state diagram below represents the language specified by the regular expression ab^*a discussed above. Since regular languages can be infinite (indeed, most interesting ones are), our state diagrams will now allow for cycles between states (or self-loops, as shown on state S2 below). To make these state diagrams efficient to implement, there is a certain restriction we must follow that may not be obvious at first. For each state, there is always only one transition on a given symbol, i.e., it is incorrect to define transitions from a state on the same symbol to more than one state. In other words, transitions through the machine are deterministic; there is no choice. These state machines are called Deterministic Finite Automata (DFA).



As discussed earlier, in CS241 we make our state diagrams simpler by assuming **implicit error states**. If a state does not define a transition on a symbol, we assume that symbol transitions the state to an error state. The state machine on the right represents the same language as the one above but with an explicit error state. Notice how transitions out of the ERR state lead only to the ERR state.



Definition 6 A **DFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

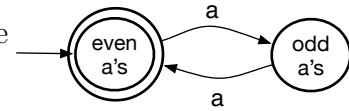
- Σ is a finite non-empty set (alphabet).
- Q is a finite non-empty set of states.
- $q_0 \in Q$ is a start state.
- $A \subseteq Q$ is a set of accepting states.
- $\delta: (Q \times \Sigma) \rightarrow Q$ is our [total] transition function (given a state and a symbol of our alphabet, what state should we go to?).

(δ is the Greek letter delta.)

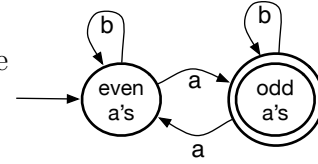
For our CS241 DFA for $L = ab^*a$, $\Sigma = \{a,b\}$, $Q = \{S1, S2, S3\}$, $q_0 = S1$, $A = \{S3\}$, and $\delta(S1, a) = S2$, $\delta(S2, b) = S2$ and $\delta(S2, a) = S3$ ⁶.

⁶Notice that the simplification we make in drawing DFAs in CS241 carries (to some extent) to our formal definition of a DFA. Technically speaking, for this example $Q = \{S1, S2, S3, ERR\}$ where *ERR* is the error state we choose not to draw in CS241 (and many others do the same). Similarly, the transition function is total, i.e., it must define a transition from each state in Q on each symbol in Σ . Yet we have undefined transitions in our example, e.g., $\delta(S1, b)$ is undefined. We assume that for all states $q \in Q$ which do not define a transition for a symbol $w \in \Sigma$, there is a transition $\delta(q, w) = ERR$.

Example: Draw a DFA for a language over $\{a\}$ where words in the language have an even number of as.



Example: Draw a DFA for a language over $\{a,b\}$ where words in the language have an odd number of as.

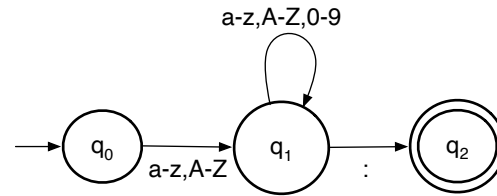


Draw a DFA for a language over $\{0,1\}$ where words in the language have an even length.

Example: Labels in MIPS form a regular language. Represent this language as a DFA.

- $\Sigma = \{a.....z, A.....Z, 0.....9, :\}$.
- $Q = \{q_0, q_1, q_2\}$.
- q_0 is our start state.
- $A = \{q_2\}$. (Note: this is a set!)
- δ is defined by

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



3.3.1 DFA recognition algorithm

We can define a recognition algorithm for words in a regular language using a DFA. For this, we first extend the definition of $\delta: (Q \times \Sigma) \rightarrow Q$ recursively to a function $\delta^*: (Q \times \Sigma^*) \rightarrow Q$ via:

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

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

In words, while δ defined a single transition for a given character, δ^* defines a sequence of transitions based on an input string and provides the resulting state after the sequence of characters in the string has been consumed.

⁷The answer is found in Section 7.1 at the end of this module.

Definition 7 A DFA given by $M = (\Sigma, Q, q_0, A, \delta)$ **accepts a string** w if and only if $\delta^*(q_0, w) \in A$.

In other words, to recognize a word $w = a_1a_2...a_n$, we start at the DFA's start state q_0 and transition on the first symbol a_1 to end up in some state q_i . We then transition from q_i on symbol a_2 . We repeat until we have transitioned on all symbols, a_1 through a_n . At that point, if the resulting state is accepting, we accept (recognize). Otherwise, reject.

Algorithm 2 DFA Recognition Algorithm

```

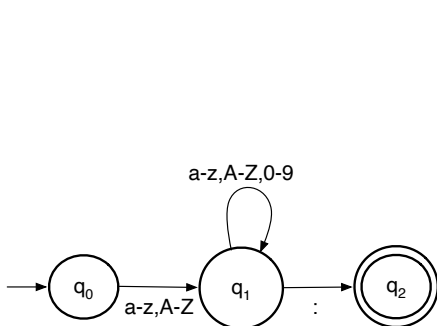
1:  $W = a_1a_2...a_n$ 
2:  $s = q_0$ 
3: for  $i$  in 1 to  $n$  do
4:    $s = \delta(s, a_i)$ 
5: end for
6: if  $s \in A$  then
7:   Accept
8: else
9:   Reject
10: end if

```

The transition function, δ can be hard-coded as a sequence of **if** statements (or a switch statement). Alternately, we could use a lookup table.

The above algorithm assumes that transitions are defined for each combination of state and symbol (which is the case when using explicit error states). We can extend this algorithm to handle the CS241 simplification of using implicit error states: if a transition is not defined from a given state on a particular symbol, we reject right away. Note that we can't do the same for accepting: we only check if we're in an accepting state after reading the entire input. It isn't unusual to transfer through an accepting state but end up in a rejecting state.

We use the DFA for MIPS labels presented earlier with two example input words to trace through the recognition algorithm:



Input is **loop2**:

Seen	Remaining	s
ϵ	loop2:	q_0
l	oop2:	q_1
lo	op2:	q_1
loo	p2:	q_1
loop	2:	q_1
loop2	:	q_1
loop2:	ϵ	q_2

Accept, as the algorithm ends at an accepting state.

Input is **end\$**:

Seen	Remaining	s
ϵ	end\$:	q_0
e	nd\$:	q_1
en	d\$:	q_1
end	\$:	q_1

Reject, as there is no transition defined from q_1 on symbol \$.

A video illustration of how [DFA recognition](#) works is available.

Using this recognition algorithm, we can define the language of a DFA as:

Definition 8 The *language of a DFA* M , denoted $L(M)$, is the set of all strings accepted by M , that is:

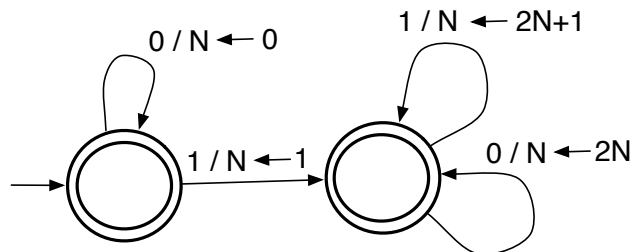
$$L(M) = \{w : M \text{ accepts } w\}$$

One thing we have taken for granted so far is that a DFA represents a regular language. In CS360/365 you will prove the following theorem:

Theorem 1 (Kleene) 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.

3.3.2 Extension to DFAs

We can extend DFAs by attaching actions to each transition. For example, consider the language of binary numbers shown as a DFA below. For each transition, we have added an action to perform while taking that transition. Such DFAs are called finite transducers.



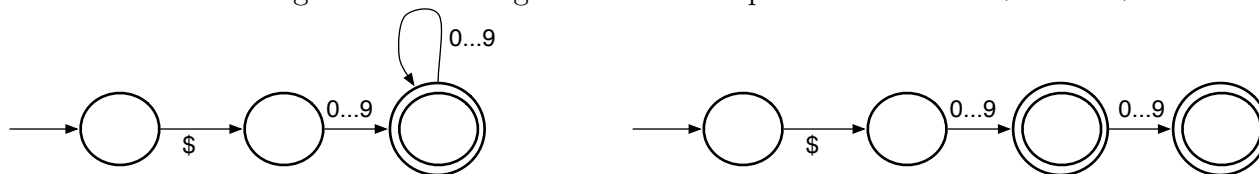
What do the actions in DFA above do?

Consider a word 1011. We transition from the start state on symbol 1. At the same time, we perform that action setting N to 1. We then transition on symbol 0. This causes N to be set to $2N$, i.e., 2. We transition on 1, setting N to $2N+1$, 5. We transition on the last 1 which sets N to 11. This gives us the decimal representation of the binary number, 1011.

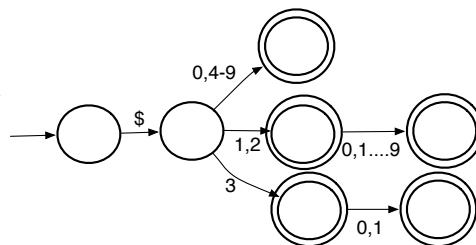
3.3.3 DFAs and compilers

DFAs (sometimes with actions) are used in the Scanning stage of a compiler, where the input is a sequence of characters representing the program, and the output is a sequence of tokens.

As an example of using DFAs to specify the words to be recognized as tokens, the two DFAs below attempt to specify the regular language where words are valid registers in MIPS. The DFA on the left allows words such as \$01234, which are not valid registers. The DFA on the right is only slightly better. It restricts registers to two digits but still accepts words such as \$00 and \$89.



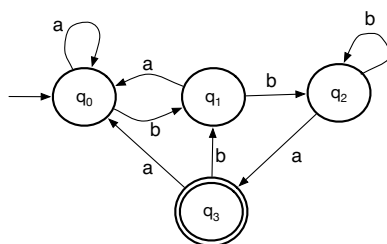
The DFA to the right correctly represents exactly the language of valid register values in a MIPS Assembly program. Notice the trade-off between precisely defining the set of valid registers or allowing some invalid words to be considered valid. This trade-off occurs in compilers during the scanning stage.



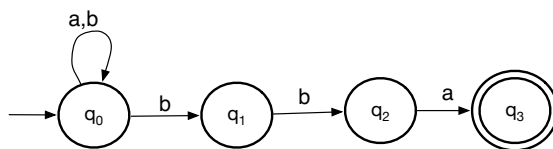
Compilers will often tokenize using a simpler DFA and then programmatically weed out invalid words. For example, the compiler could use the three-state DFA (above and to the left). Of course, this DFA accepts strings such as \$01234 as valid registers even though they are not. The compiler can enforce a range check after tokenization to only approve register values that are within the legal range of 0 and 31.

3.4 Non-Deterministic Finite Automata

Consider the DFA for the language $L = \{w : w \text{ ends with } bba\}$ over the alphabet $\Sigma = \{a, b\}$.



What if we were willing to allow more than one transition from a state on the **same** symbol? The machine would then be able to **choose** which path to go on. This is called **non-determinism**. With such a machine, the machine would accept a word w if and only if there exists some path that leads to an accepting state. Using this, we could simplify the DFA above and represent it as the following Non-Deterministic Finite Automata (NFA):



Given an input word, the machine chooses to stay at q_0 until it sees the ending bba . Of course, computers are deterministic, so implementing non-determinism may come at a performance cost. The computer can only try one non-deterministic choice at a time, and in the worst case it might have to try every possibility! Nonetheless, this idea is useful because it gives us flexibility in representing languages.

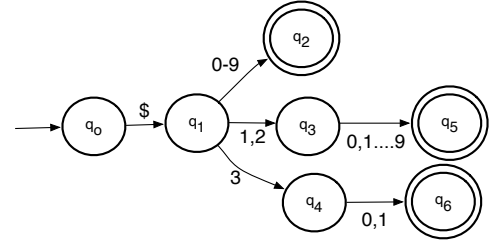
To formalize the recognition algorithm, let's define an NFA and languages defined by NFAs.

Definition 9 An **NFA** is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ is a finite non-empty set (alphabet).
- Q is a finite non-empty set of states.
- $q_0 \in Q$ is a start state.
- $A \subseteq Q$ is a set of accepting states.
- $\delta: (Q \times \Sigma) \rightarrow 2^Q$ is our [total] transition function. Note that 2^Q denotes the power set of Q , that is, the set of all subsets of Q .

The only difference between a DFA and an NFA is how the transition function is defined. In a DFA, we had defined the transition function to take a state and a symbol and result in a single new state. In an NFA, a state and symbol combination can now lead to a **set** of states.

In Section 3.3.3, we discussed the DFA for recognizing valid registers in MIPS. The same regular language can be represented by the NFA on the right. Notice how we no longer have to specify transitions on symbols 0 and then 4 through 9 separately and can simply define one transition on 0 through 9 to state q_2 and, at the same time, define a transition on symbols 1 and 2 to state q_3 and another on symbol 3 to state q_4 .



3.4.1 NFA Recognition Algorithm

The transition function for an NFA allows us to transition from one state to multiple states consuming a single symbol. Like before, we can extend this to multiple symbols, i.e., we can extend the definition of $\delta: (Q \times \Sigma) \rightarrow 2^Q$ to a function $\delta^*: (2^Q \times \Sigma^*) \rightarrow 2^Q$ via:

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

where $a \in \Sigma$.

Analogously, we also have:

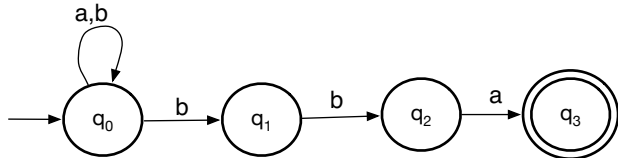
Definition 10 An NFA given by $M = (\Sigma, Q, q_0, A, \delta)$ **accepts a string** w if and only if $\delta^*({q_0}, w) \cap A \neq \emptyset$.

In other words, to recognize a word, $w = a_1a_2...a_n$, we start at the NFA’s start state q_0 and transition on the first symbol a_1 to end up in a set of states S_i . We then transition on each state q_i from S_i on symbol a_2 to a new set of states. We repeat until we have transitioned on all symbols, a_1 through a_n . At that point, if the resulting set of states contains an accepting state, we accept (recognize). Otherwise, reject ⁸.

Algorithm 3 NFA Recognition Algorithm

```
1:  $w = a_1a_2...a_n$ 
2:  $S = \{q_0\}$ 
3: for  $i$  in 1 to  $n$  do
4:    $S = \bigcup_{q \in S} \delta(q, a_i)$ 
5: end for
6: if  $S \cap A \neq \emptyset$  then
7:   Accept
8: else
9:   Reject
10: end if
```

As a concrete example, let’s once again consider the NFA from above, which is reproduced below along with a trace of the recognition algorithm using the input word *abbba*:



Seen	Remaining	S
ϵ	<i>abbba</i>	$\{q_0\}$
<i>a</i>	<i>bbba</i>	$\{q_0\}$
<i>ab</i>	<i>bba</i>	$\{q_0, q_1\}$
<i>abb</i>	<i>ba</i>	$\{q_0, q_1, q_2\}$
<i>abbb</i>	<i>a</i>	$\{q_0, q_1, q_2\}$
<i>abbba</i>	ϵ	$\{q_0, q_3\}$

Once the entire word has passed through the NFA, S is $\{q_0, q_3\}$. Since this contains the accepting state q_3 , the word is accepted.

A video illustration of how [NFA recognition](#) works has been uploaded.

Similar to before, we have the following definition:

Definition 11 Denote the **language of an NFA** M to be the set of all strings accepted by M , that is:

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

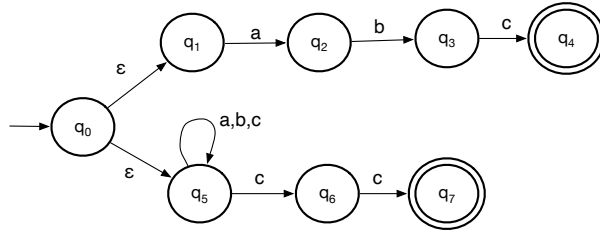
Previously, we discussed how every language that can be represented as a DFA is regular (Kleene’s Theorem). The same is true for languages that can be represented as an NFA. In fact, every NFA can be converted to a DFA that recognizes the same language. CS 360/365 address this in more detail. We have sketched some details in the Appendix (Section 6). This is considered optional reading for the course.

⁸An alternate, “board game style” interpretation of NFA recognition goes as follows: you start with one piece in the start state. When you read a symbol, you remove your piece from the current state, and place new pieces in each of the states you transition to on the corresponding symbol. In each turn, you do this for all the states you are currently in. You “win the game” (accept) if one of your pieces is in an accepting state after reading the whole word.

3.5 ϵ -Non-Deterministic Finite Automata

Another generalization to DFAs/NFAs is to permit state transitions without consuming a symbol. These are known as ϵ transitions and the resulting machines are called ϵ -Non-Deterministic Finite Automata (ϵ -NFA). One use of ϵ -transitions is to take the union of two NFAs by “gluing” the NFAs. While this is possible with ordinary NFAs, ϵ -transitions make it more straightforward. We will see later that ϵ -transitions also make it easy to construct the concatenation of two NFAs and the Kleene star of an NFA.

For example, for the language $L = \{abc\} \cup \{w : w \text{ ends with } cc\}$, we can create an ϵ -NFA by creating a new start state and connecting it to the NFAs for $\{abc\}$ and $\{w : w \text{ ends with } cc\}$ using ϵ transitions:



Definition 12 An ϵ -NFA is a 5-tuple $(\Sigma, Q, q_0, A, \delta)$:

- Σ is a finite non-empty set (alphabet) **that does not contain the symbol ϵ** .
- Q is a finite non-empty set of states.
- $q_0 \in Q$ is a start state.
- $A \subseteq Q$ is a set of accepting states.
- $\delta: (Q \times \Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is our [total] transition function. Note that 2^Q denotes the power set of Q , that is, the set of all subsets of Q .

3.5.1 ϵ -NFA Recognition Algorithm

Definition 13 The epsilon closure, $E(S)$, of a set of states S , is the set of all states reachable from S in 0 or more ϵ -transitions.

Note this implies that $S \subseteq E(S)$. As an example, for the ϵ -NFA above, $E(\{q_0\}) = \{q_0, q_1, q_5\}$.

We can extend the definition of $\delta: (Q \times \Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ to a function $\delta^*: (2^Q \times \Sigma^*) \rightarrow 2^Q$ via:

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

where $a \in \Sigma$. Analogously, we also have:

Definition 14 An ϵ -NFA given by $M = (\Sigma, Q, q_0, A, \delta)$ **accepts a string w** if and only if $\delta^*(E\{q_0\}, w) \cap A \neq \emptyset$.

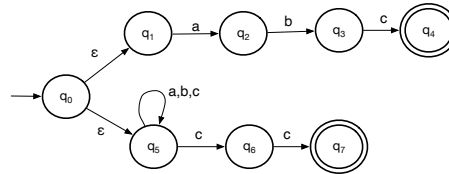
Recognition of a word using an ϵ -NFA follows the same algorithm as recognizing a word using an NFA with the one difference that for each set of states (including the starting state), we compute its epsilon closure.

Algorithm 4 ϵ -NFA Recognition Algorithm

```

1:  $w = a_1a_2...a_n$ 
2:  $S = E(\{q_0\})$ 
3: for  $i$  in 1 to  $n$  do
4:    $S = E(\cup_{q \in S} \delta(q, a_i))$ 
5: end for
6: if  $S \cap A \neq \emptyset$  then
7:   Accept
8: else
9:   Reject
10: end if
  
```

As a concrete example, we trace two inputs using the recognition algorithm on our previous ϵ -NFA example that is reproduced below:



Trace for the input word *abcaccc*:

Seen	Remaining	S
ϵ	<i>abcaccc</i>	$\{q_0, q_1, q_5\}$
<i>a</i>	<i>bcaccc</i>	$\{q_2, q_5\}$
<i>ab</i>	<i>caccc</i>	$\{q_3, q_5\}$
<i>abc</i>	<i>accc</i>	$\{q_4, q_5, q_6\}$
<i>abca</i>	<i>ccc</i>	$\{q_5\}$
<i>abcac</i>	<i>cc</i>	$\{q_5, q_6\}$
<i>abcacc</i>	<i>c</i>	$\{q_5, q_6, q_7\}$
<i>abcaccc</i>	ϵ	$\{q_5, q_6, q_7\}$

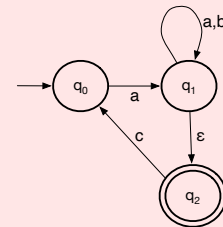
Since $\{q_5, q_6, q_7\} \cap \{q_4, q_7\} \neq \emptyset$, accept *abcaccc*.

Trace for the input word *abcac*:

Seen	Remaining	S
ϵ	<i>abcac</i>	$\{q_0, q_1, q_5\}$
<i>a</i>	<i>bcac</i>	$\{q_2, q_5\}$
<i>ab</i>	<i>cac</i>	$\{q_3, q_5\}$
<i>abc</i>	<i>ac</i>	$\{q_4, q_5, q_6\}$
<i>abca</i>	<i>c</i>	$\{q_5\}$
<i>abcac</i>	ϵ	$\{q_5, q_6\}$

Since $\{q_5, q_6\} \cap \{q_4, q_7\} = \emptyset$, reject *abcac*.

For the ϵ -NFA presented to the right and the input word *abca*, provide a step by step trace through the ϵ -NFA recognition algorithm. Is the input accepted? Why or why not?



9

Every ϵ -NFA can be converted to an NFA. This fact is often proven as part of proving Kleene's theorem. We sketch the details of such a conversion in the Appendix to this module (Section 6).

⁹The answer is found in Section 7.2 at the end of this module.

Since every ϵ -NFA can be converted to an NFA, and every NFA can be converted to a DFA, every ϵ -NFA can be converted to a DFA. This allows us to use the flexibility of ϵ -NFAs to build a recognizer, convert to DFA, then use the determinism of the DFA recognition algorithm to recognize the language efficiently; in fact, this is precisely how tools like `egrep` work.

4 Scanning

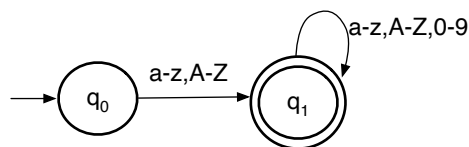
In the module on Machine Languages, we discussed writing an assembler for the MIPS Assembly language. We talked about the first step in this process, the conversion of a sequence of characters into tokens. This is the Scanning stage.

The obvious first question to ask is whether the words in a programming language form a regular language. In the previous sections, we already discussed how MIPS labels and MIPS register values are regular (we represented them using DFAs). What about a higher-level language like C? The smallest components of a C program, such as keywords (`if`, `while` etc.), identifiers (function names, variable names etc.), literals (`42`, `0x15`, `"Hello World"` etc.), operators (`+`, `++`, `+=` etc.) and comments are indeed regular. Finite automata could indeed be used for recognition.

This module has discussed recognition as a decision algorithm that accepts or rejects an input depending on whether it is a **single** word in the language. However, recognition is not the same as scanning. While recognition determines whether a string is a single word in the language, scanning is the process of taking a string and breaking it into words that are in the language, i.e., given an input string s , the result of scanning this string is either a scanning error or a sequence of words w_1 through w_n where the concatenation of the words yields s .

One can visualize how a scanning algorithm could be created by leveraging the ability to recognize individual words in a language. Given the string s , the recognition algorithm would be run on a DFA for the language using s as input. At some point, the DFA would get stuck (more on this later). If the DFA is stuck on an accepting state, i.e., a word has been found, a token corresponding to that word would be emitted. Then the scanning algorithm would reset the recognition DFA to the start state. This process is repeated on the still unseen part of s until either all of s has been consumed and we have our tokens, or the input has been rejected.

While this looks straightforward, there is one issue we must resolve. Let's take a concrete example by considering the language L of just ID (identifier) tokens in C and using the following DFA ¹⁰:



Now consider the input `abcde`. How should this be tokenized? On one extreme, we could tokenize the input as a single ID token representing the entire string. On the other extreme, we could generate five different ID tokens, each with a string containing just one character. From an implementation perspective, the question to answer is whether we should emit a token as soon as the recognition

¹⁰We make the assumption that identifiers cannot contain underscore which is technically incorrect. However, this does not take anything away from our discussion.

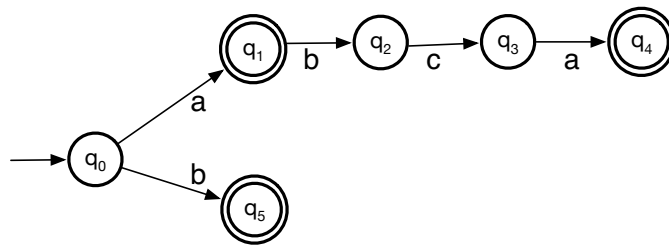
algorithm reaches an accept state, or whether we should continue onwards (if a valid transition is available) until we find no transitions or the input is exhausted?

Other components of C have a similar issue. For example, the input string `0x12CC` could be a single HEXINT or could be an INT (0) followed by an ID (x) followed by another INT (1) followed by another INT (2) and followed by an ID (CC). Which interpretation is correct? Which one do compilers use?

4.1 Maximal Munch Scanning Algorithm

Scanners typically use flavours of the maximal munch scanning algorithm, an algorithm that belongs to a class of algorithms called *greedy algorithms*. The algorithm is termed greedy as at each stage it attempts to consume the maximum number of characters it can. For the language of identifiers above, the maximal munch algorithm would generate a single token, an ID with the string `abcde`. Similarly, for the language of all C components, the input `0x12CC` would be read as a single HEXINT token.

To illustrate the algorithm, let's consider the following example with $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$ and the DFA as shown below:

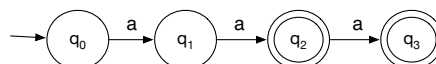


Consider the input string, $s = ababca$. The algorithm consumes a and **flags this state, q_1 , as it is accepting**. Being greedy, the algorithm continues to consume more input rather than outputting this token. The algorithm consumes b and reaches state q_2 . At this point, the algorithm is stuck; it is not at an accepting state and there is no transition on symbol a (the next symbol in the input). The algorithm **backtracks** to the **last seen accepting state**, “un-consuming” input that it had greedily consumed. The last seen accepting state is q_1 , when only a had been consumed. The algorithm outputs an a token and resets the state to q_0 , the start state. The algorithm then resumes consuming b and flags state q_5 as the last seen accepting state. At this point, the algorithm is stuck again as there is no transition on a . Since the current state is an accepting state, the algorithm outputs token b and resets to q_0 . The algorithm then consumes the second a , the second b , the first c , the third a and runs out of input. This last state, q_4 , is accepting so the last token $abca$ is output.

What would this algorithm output for the input string `baba` on the above DFA?

11

The algorithm presented isn't without its own problems. Consider the language $L = \{aa, aaa\}$ and the corresponding scanning machine:



¹¹The algorithm would output a b token, followed by an a token, then another b token and finally another a token.

If $s = aaaaa$, maximal munch will output the token aaa followed by aa .

Now consider the input $s = aaaa$. With the greedy approach, the algorithm will consume aaa and reach q_3 . At this point, it is stuck at an accepting state, so it outputs the token aaa and the state is reset to q_0 . The algorithm then consumes the last remaining a and is stuck at state q_1 . This is not an accepting state and therefore the algorithm rejects the tokenization of the input $aaaa$. Of course, a valid tokenization (aa followed by aa) is possible. The maximal munch algorithm isn't guaranteed to find a tokenization if a tokenization exists.

The maximal munch algorithm, as presented above, has quadratic time complexity due to backtracking. A more complex implementation that uses memoization and dynamic programming is available that gives linear time complexity.

Can you think of a language and input that would demonstrate this quadratic time complexity?

¹²

4.1.1 Maximal munch in real life

Consider the following Java code:

```
int i=1;
int j=2;
System.out.println(i+++j);
```

This code is legal Java code and produces the output of 3 with the value of i and j both 2 after the print statement. Now that we know how maximal munch works, we can deduce the tokenization. The expression $i+++j$ (notice that lack of any whitespace) is tokenized as i , $++$, $+$ and j . This means that the expression $i++$ is executed, i.e., the post increment operator (take the old value of i for the expression) which produces the output we witness.

Notice that an expression of the form $i+++++j$ (no whitespace) will lead to the tokenization: i , $++$, $++$, $+$ and j . This will lead the compiler to generate a syntax error. However, had we used whitespace $i++ + ++j$ or even $i+++ ++j$, the resulting tokenization would be i , $++$, $+$, $++$ and j which is valid syntax.

Let's consider a famous (though now obsolete) example from C++:

```
vector<pair<string,int>>> v;
```

In C++03 or earlier, this caused a compiler error. The reason was that part of the input contained $>>$, which is a valid operator (right bit shift or the output operator). Due to this, since maximal munch is greedy, it outputs a single $>>$ token rather than two $>$ tokens that the syntax expects. In earlier versions of C++, the only way around it was to place whitespace in between the two $>$ symbols. As of C++11, C++ does what Java did all along; a post-tokenization stage checks for instances such as this and changes the tokenization.

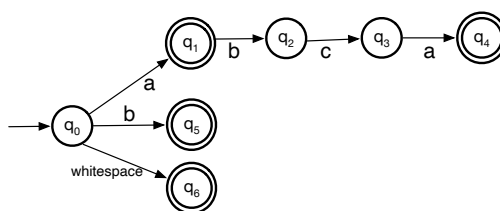
¹²The answer is found in Section 7.3 at the end of this module.

4.2 Simplified Maximal Munch Algorithm

As can be seen from the previous examples, one common solution to limitations of the maximal munch algorithm is to have the programmer explicitly separate tokens by using whitespace. With this work-around in place, maximal munch works well in practice. We present a simplification to the maximal munch algorithm that we name *simplified maximal munch*. Our simplification is to not backtrack at all, i.e., while being greedy, the algorithm continues to consume input until it gets stuck. At that point, if the algorithm is at an accepting state, it will produce a token. Otherwise, the algorithm will **not** backtrack and will reject the input and declare it un-tokenizable. The advantage of this simplification is that by removing backtracking, a straightforward implementation of the algorithm has linear time complexity in the length of the input being scanned.

Let's once again look at the same example that we used for illustrating the maximal munch algorithm: $\Sigma = \{a, b, c\}$, $L = \{a, b, abca\}$, $s = ababca$. The algorithm begins greedily and consumes a followed by b . At this point it is stuck. Since the algorithm is not at an accepting state (as $ab \notin L$) and does not do any backtracking, the algorithm rejects the input. In other words, the algorithm has rejected $ababca$ even though we know that a valid tokenization is possible through the un-simplified maximal munch algorithm. In practice, the algorithm is usually good enough, and the known limitations can be avoided by adding whitespace as a way to indicate the intended tokenization. For example, we can change the language to accommodate for whitespace as shown in the diagram below.

The input $w = a\ b\ abca$ (note the whitespace) is tokenized correctly by simplified maximal munch using the modified language. The algorithm gets stuck at state q_1 and emits the token a and resets to q_0 . Since the next symbol is whitespace, the algorithm transitions to state q_6 , gets stuck on encountering the symbol b , emits a whitespace token and resets to q_0 . The algorithm will continue on to emit a b token followed by a whitespace token, and then an $abca$ token. If the whitespace tokens are not needed, a post-processing step could filter out these tokens.



Is it possible that a string can be scanned into tokens, but both simplified maximal munch and ordinary maximal munch reject the string?

13

A video illustration of how [maximal munch](#) works has been uploaded.

¹³Yes. We actually saw this earlier with the DFA that recognizes the language $\{aa,aaa\}$. Maximal munch was unable to tokenize the string $aaaa$, even though the tokens aa and aa are a valid tokenization. Since simplified maximal munch is less powerful, it has the same limitation.

4.3 Algorithm for Simplified Maximal Munch

The algorithm assumes that the input machine is a DFA $(\Sigma, Q, q_0, A, \delta)$. The algorithm assumes the function $\text{peek}(w)$ which returns the next symbol in the input string without consuming it and the function $\text{consume}(w)$ which consumes and returns the next symbol from the input. For example, for an input string $w = a_1a_2\dots a_n$, $\text{peek}(w)$ and $\text{consume}(w)$ will both return a_1 but, while $\text{peek}(w)$ leaves w unchanged, $\text{consume}(w)$ mutates the input string to become $a_2\dots a_n$.

Algorithm 5 Simplified Maximal Munch

```
1:  $w$  = input string
2:  $s = q_0$ 
3: repeat
4:   if  $\delta(s, \text{peek}(w)) == \text{ERROR}$  then
5:     if  $s \in A$  then
6:       Output token for state  $s$ 
7:        $s = q_0$ 
8:     else
9:       Reject
10:    end if
11:  else
12:     $s = \delta(s, \text{consume}(w))$ 
13:  end if
14: until  $w$  is empty
15: if  $s \in A$  then
16:   Output token for state  $s$  and Accept
17: else
18:   Reject
19: end if
```

5 Looking ahead

In this module, we learned about a class of languages called regular languages. While we formally defined regular languages, our goal was to leverage the power of regular languages to represent the smallest components of higher-level programs. We have seen how regular languages can be used to break an input program into a sequence of tokens, i.e., implement a scanner. In the next module, we talk about another class of languages, Context-Free Grammars, and then discuss how we can take the output of a scanner to construct a representation of the program that we are trying to compile, i.e., write a parser.

This module introduced you to regular languages. One thing that you have not learned, and will not learn in this course, are the tools necessary to formally prove when a language is **not** regular. But, based on what you know about finite automata and regular expressions, can you think of a language that intuitively does not seem to be regular, i.e., cannot be represented as a finite automata or as a regular expression.

14

¹⁴An answer will be provided at the start of the next module.

6 Appendix: Kleene's Theorem

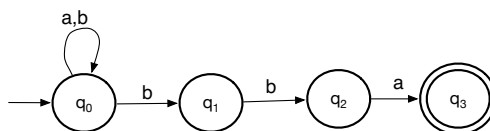
This is an appendix to the module. Readers can read this material if it interests them. It is **NOT** considered required reading.

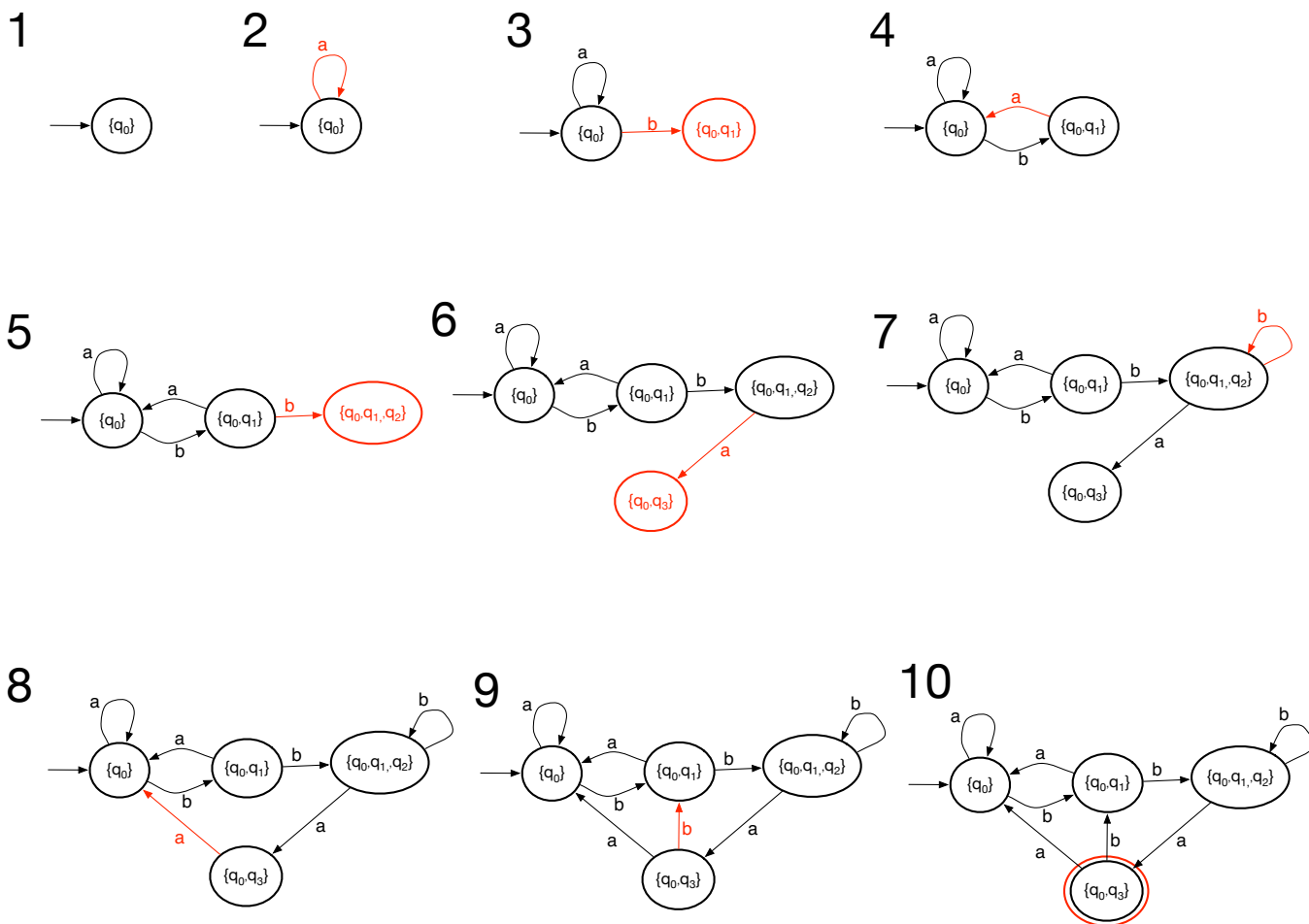
6.1 NFA to DFA conversion

Every NFA can be converted to a DFA that recognizes the same language. The basic idea is to take advantage of the facts that there are only finitely many sets of states that can be reached in an NFA, and that we can use any name for our states. We will construct a DFA where the states are named after sets of states from the NFA and connect them using the NFA's transition function. Each state in the DFA will correspond to the set of states that we would have reached after seeing some part of the input had we been using the NFA. To do this, we could write down all of the 2^Q possible sets of states and then connect them one by one based on δ^* and each letter in Σ . This however leads to a lot of extra states and a lot of unnecessary work. Instead, we use an algorithm called subset construction:

- Start with the state $S = \{q_0\}$.
- Using the NFA, determine what happens for each $q \in S$ separately for each symbol $a \in \Sigma$. The set of resulting states becomes its own state in our DFA, and a transition is added from state S to this new state on symbol a .
- Repeat the previous step for each new state created until we have exhausted every possibility.
- Accepting states are any states that included an accepting state of the original NFA.

On the next page, we show each step of this algorithm and convert the following NFA to a DFA.



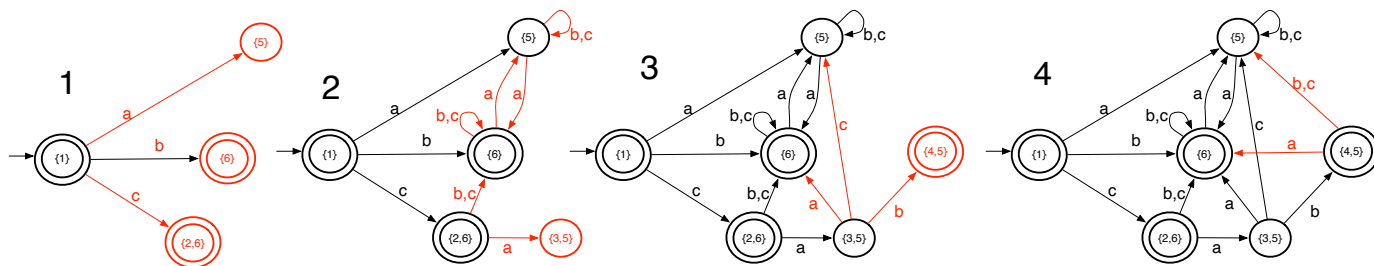
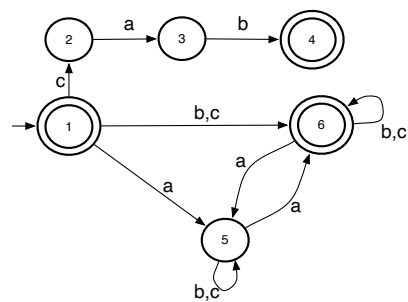


We demonstrate [NFA to DFA construction](#) in a video.

We illustrate the construction with another example (this time skipping some steps) using the language:

$$L = \{cab\} \cup \{w : w \text{ contains even number of } a\text{'s}\}$$

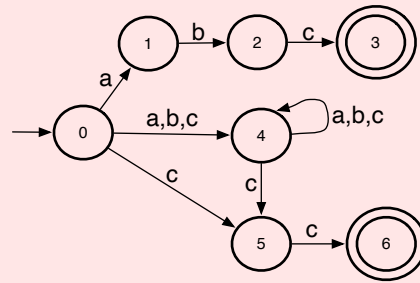
The NFA for this language is shown on the right.



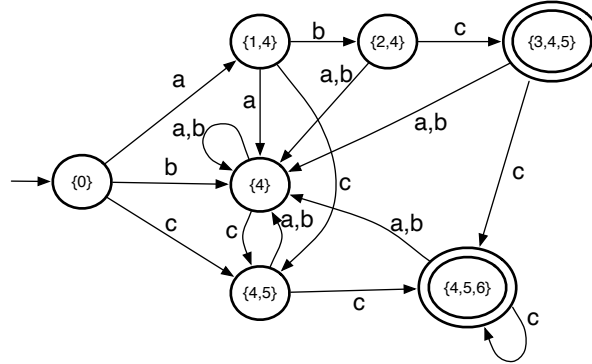
Using subset construction, convert the following NFA for the language:

$$L = \{abc\} \cup \{w : w \text{ ends with } cc\}$$

into a DFA.



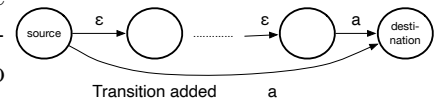
Solution:



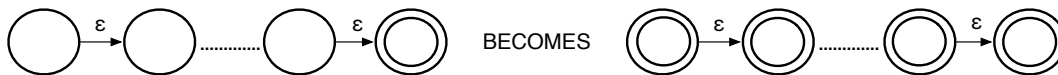
6.2 ϵ -NFA to NFA Conversion

There is a simple algorithm that can convert any ϵ -NFA to an NFA. We informally discuss the algorithm below:

1. If a transition path from **source** state to a **destination** state consists of a sequence of ϵ transitions followed by a single transition on symbol a , add a direct transition from **source** to **destination** labeled with symbol a .

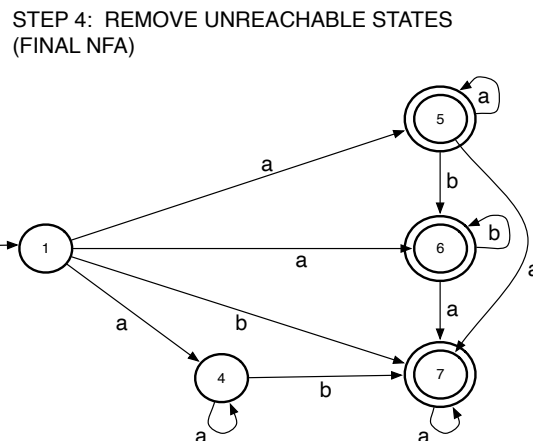
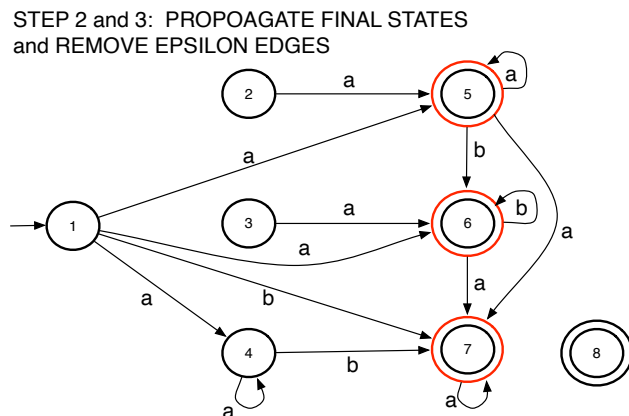
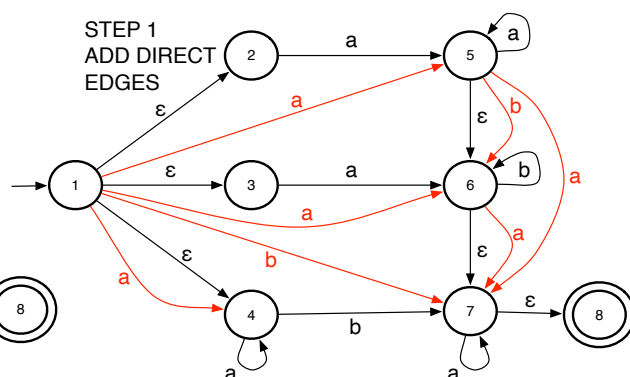
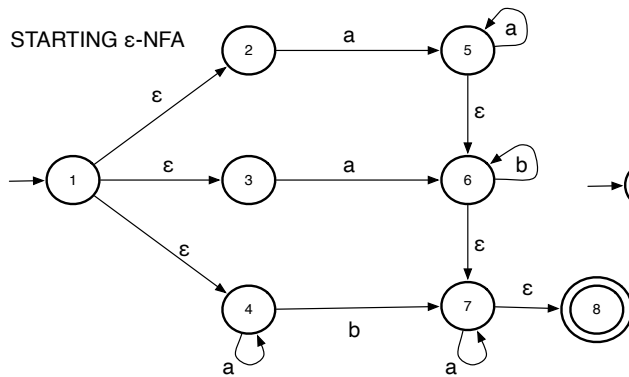


2. If a sequence of ϵ transitions leads to an accepting state, make all states in the sequence accepting.



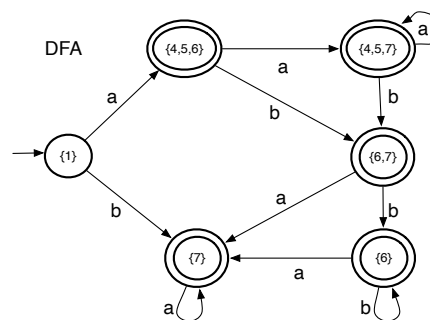
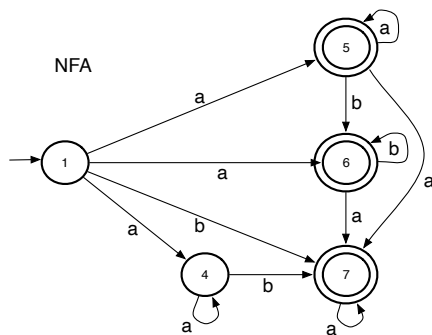
3. Remove all ϵ transitions.
4. Remove all unreachable states.

Below is a concrete example of this construction.



Take the FINAL NFA created above and convert it to a DFA using subset construction.

Solution:

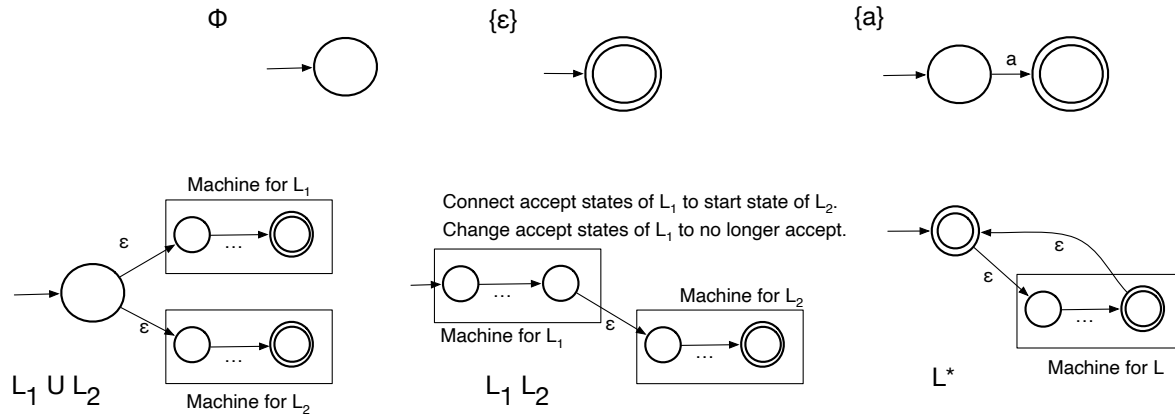


The conversion of an ϵ -NFA to an NFA (or DFA) combined with Kleene's Theorem implies that every language recognized by an ϵ -NFA is regular.

6.3 Kleene's Theorem

We have discussed Regular Expressions, DFAs, NFAs and ϵ -NFAs as four ways to represent regular languages. We demonstrated ways to convert an NFA to a DFA and an ϵ -NFA to an NFA. If we were to show that any regular expression can be converted to an ϵ -NFA, then that would complete one direction of Kleene's Theorem (CS360/365 cover both directions). Below we show informally how to construct ϵ -NFAs for each of the cases in the inductive definition of regular languages.

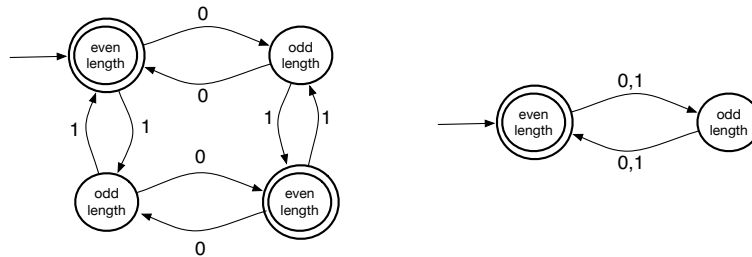
One can make this argument formal using structural induction. The diagram below shows how to construct DFAs for each of the base cases in the definition of regular languages. Then, for each of the recursive cases, suppose we have automata for two regular languages (or one language in the case of star). The diagram shows how to combine or extend these automata using epsilon transitions, to create automata for the union, concatenation or star. Therefore, we can repeatedly apply these constructions to create an e-NFA for any regular language.



7 Answers to longer self-practice questions

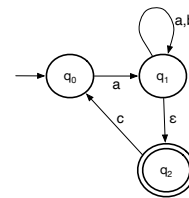
7.1 Self-practice answer 1

Draw a DFA for a language over $\{0,1\}$ where words in the language have an even length.



7.2 Self-practice answer 2

For the ϵ -NFA presented below and the input word *abca*, provide a step-by-step trace through the ϵ -NFA recognition algorithm. Is the input accepted? Why or why not?



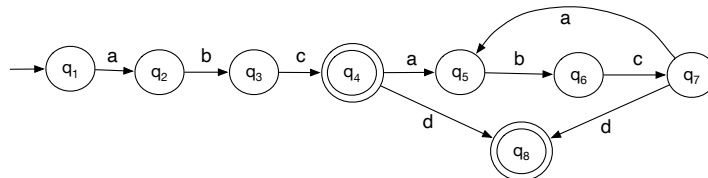
Processed	Remaining	S
ϵ	<i>abca</i>	$\{q_0\}$
<i>a</i>	<i>bca</i>	$\{q_1, q_2\}$
<i>ab</i>	<i>ca</i>	$\{q_1, q_2\}$
<i>abc</i>	<i>a</i>	$\{q_0\}$
<i>abca</i>	ϵ	$\{q_1, q_2\}$

Since $\{q_1, q_2\} \cap \{q_2\} \neq \emptyset$, accept.

7.3 Self-practice answer 3

Can you think of a language and input that would demonstrate this quadratic time complexity?

Consider the regular language $abc|abc(abc)^*d$, also shown as a DFA below:



Consider the input *abcabcabcabcabcabcabcabc*. A valid tokenization for this is a sequence of *abc* tokens. However, being greedy, maximal munch will keep reading the input word until it gets stuck at the very end of input (since it will not see a terminating *d*). It will then backtrack to the last seen accepting state which was encountered after having read the first *abc*. The algorithm would then restart the entire process and get stuck once again at the very end. It will then backtrack to the last seen accepting state which is the second occurrence of *abc*.