Modeling Computation

R Pruim

2019-05-08

Contents

Preface

These exercises were assembled to accompany a Discrete Mathematics course at Calvin College.

1 Alphabets, Languages, and Grammars

1.1 Terminology

- An **alphabet** (or **vocabulary**) is just a finite, non-empty set. It's elements are called **letters** or **symbols**.
- A **word** (or **string**) is a finite sequence of symbols.
- A word consisting of no symbols is called the **empty word** and denoted λ. (Think `""`.)
- The set of all words using exactly n symbols from V (repetition allowed) is denoted V^n.
- The set of all words using symbols in alphabet V is denoted V^*. (So $V^* = V^0 \cup V^1 \cup V^2 \cdots$.)
- A **language** over V is a subset of V^*.

Exercises

1. Let $V = \{0, 1\}$. What is V^0? What is V^2? What is V^*?
2. Let $V = \{2\}$. What is V^0? What is V^2? What is V^*?
3. Can $\emptyset$ (empty set) be a letter? an alphabet? a word? a language?

1.2 (Phrase-Structure) Grammars

A phase-structure grammar consists of

- **alphabet**: V
- **start symbol**: $S \in V$
- **terminal symbols**: $T \subset V$
  - $N = V - T$ is the set of **nonterminal symbols**
- **production rules**: $P \subseteq (V^* - N^*) \times V^*$
  - usually write elements of P as $u \to v$ rather than (u, v).
  - $V^* - N^*$ says that u must contain at least on nonterminal.
  - v can be any combinaton of terminals and nonterminals (including the empty string).

Examples

Note: in each of these examples, capital letters are used for nonterminals and lower case letters or digits for terminals. That makes it easy to remember, but it is not a requirement of the definition of a grammar.

Grammar 1 (G_1): alphabet: $\{a, b, A, B, S\}$, terminals: $\{a, b\}$, start symbol: S, production rules:

- $S \to ABa$
- $A \to BB$
- $B \to ab$
- $AB \to b$

Grammar 2 (G_2): alphabet: $\{S, A, a, b\}$, terminals: $\{a, b\}$, start symbol: S, production rules:

- $S \to aA$
- $S \to b$
- $A \to aa$

Grammar 3 (G_3): alphabet: $\{S, 0, 1\}$, terminals: $\{0, 1\}$, start symbol: S, production rules:

- $S \to 11S$
- $S \to 0$

Grammar 4 (G_4): alphabet: $\{S, A, B, C, a, b, c\}$, terminals: $\{a, b, c\}$, start symbol: S, production rules:

- $S \to AB$
- $A \to Ca$
- $B \to Ba$
- $B \to Cb$
- $B \to b$
- $C \to cb$
- $C \to b$

1.2.1 Derivations and Languages

The rules of a grammar are used to derive strings of terminals (elements of T^*) as follows.

- If $w_0 \to w_1$ is a rule, then $lw_0r \Rightarrow lw_1r$ for any strings l and r.
- $a \overset{*}{\Rightarrow} b$ is defined recursively. $a \overset{*}{\Rightarrow} b$ if either
  - $a \Rightarrow b$, or
  - $\exists c\ (a \Rightarrow c \wedge c \overset{*}{\Rightarrow} b)$

Basically the production rules are "rewrite rules" that allow us to replace the left side of the rule with the right side. A derivation is a sequence of rewrites.

The language of a grammar G is denoted $L(G)$ and contains all strings of terminals that can be derived from the start symbol:

$$L(G) = \{w \in T^* \mid S \overset{*}{\Rightarrow} w\}$$

1.2.1.1 Exercises

4. Show that using Grammar 1, $ABa \overset{*}{\Rightarrow} abababa$.
5. Is $abababa \in L(G_1)$, the language generated by Grammar 1?
6. What is $L(G_2)$?
7. What is $L(G_3)$?
8. Generate several words using G_4.
9. Create a grammar G_5 such that $L(G_5) = \{0^n1^n \mid n = 0, 1, 2, ...\}$
10. Create a grammar G_6 such that $L(G_6) = \{0^n1^n \mid n \in \mathbb{Z}^+\}$
11. Create a grammar G_7 such that $L(G_7) = \{0^m1^n \mid m, n \in \mathbb{N}\}$

2 Types of Grammars

Type 0: no restrictions

Type 1 (context sensitive): only two types of rules allowed

- $lAr \to lwr$ where
  - $l, r \in V^*$
  - $A \in N$
  - $w \in V^+$ [That is, $w \neq \lambda$.]
  - shorthand: `[left][nonterminal][right]` → `[left][non-empty][right]`
- $S \to \lambda$
  - shorthand: `[start]` → `[empty string]`
  - If the rule is used, the the start symbol may not appear on the right side of any rule, so other rules become `[left][non-empty without start symbol][right]`.

Type 2 (context free): only one type of rule allowed

- $A \to w$ where
  - A is a single nonterminal symbol
  - $w \in V^*$ (so no restriction here)
- shorthand: `[nonterminal]` → `[any string]`

Type 3 (regular): Three types of rules allowed

- `[nonterminal]` → `[terminal] [non terminal]` (Example: $A \to bC$)
- `[nonterminal]` → `[terminal]` (Example: $A \to b$)
- `[start]` → `[empty string]` (Example: $S \to \lambda$)

A regular/context free/context sensitive language is a language that can be generated by a regular/context free/context sensitive grammar.

1. For each grammar we have seen so far, determine whether it is regular, context free, context sensitive. (Some grammars will be more than one. All grammars are type 0.)
2. True or false: Every grammar of type n is a grammar of type $n - 1$.

2.1 Backus-Naur Form (BNF) for Context Free Grammars

Shorthand notation for type 2 grammars.

- nonterminals denoted with $\langle\rangle$.
- $\to$ written as `::=`
- All rules with same nonterminal on left written together with the list of possible right sides separated by `|`.

Example: If we have production rules $A \to Aa$, $A \to a$, and $A \to AB$, we will write

```
<A> ::= <A>a | a | <A><B>
```

BNF for ALGOL 60 identifier

```
<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>
<letter> ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

BNF for signed integer

```
<signed integer> ::= <sign><integer>
<sign> ::= + | -
<integer> ::= <digit> | <digit><integer>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

BNF specification for Java

You can find BNF for various programming languages online. For example: https://users-cs.au.dk/amoeller/RegAut/JavaBNF.html has a BNF specification for Java.

Exercises

3. Why does BNF notation only work for context free grammars?
4. For each context free grammar we have seen, give its BNF representation.

3 Finite State Automata

A **finite state automaton** consists of the following:

- a finite set of **states**: S
- a finite **input alphabet**: I
- a **start state**: $s_0 \in S$.
- a set of **final states**: $F \subseteq S$
- a **transition function**: $f : S \times I \to S$

Example (M_0) states: $\{A, B, C, D\}$; input alphabet: $\{0, 1\}$; start state: A; final states: $\{A, D\}$; transition function described in table below

M_0								
state	A	A	B	B	C	C	D	D
letter	0	1	0	1	0	1	0	1
f	A	B	A	C	A	A	C	B

3.1 Graph Representation

1. It is often easier to visualize what is going on if we represent an automaton with a graph. Create a labeled, directed graph with the following properties:
   - There is a vertex for each state, labeled with the state. (Draw this as a circle with the state inside the circle.
   - There is an edge from state s to state t labeled with letter x if and only if $f(s, x) = t$
   - Final states are circled a second time. (We could use shape or color or something else to make it clear which states are final states, but double circling is easy.)
   - An extra arrow (not coming from any state) points to the start state. This isn't really an edge in the graph, just an extra bit of labeling.

3.2 Extended transition function

We can extend the transition function to $f^* : S \times I^* \to S$ with the following recursive definition:

- $f^*(s, \lambda) = s$ for any state s
- $f^*(s, xa) = f(f^*(s, x)a)$ for any $x \in I^*$ and $a \in I$

2. Use this definition to determine the following.
   a. $f^*(B, \lambda)$
   b. $f^*(B, 0)$
   c. $f^*(B, 010)$
   d. $f^*(A, 101)$
   e. $f^*(A, 1011)$

3.3 Language Recognition

For any automaton M with transition function f, start state s and final states F, the language recognized by M (written $L(M)$) is defined as follows:

$$x \in L(M) \iff f^*(s, x) \in F$$

3. For each string x below, compute $f^*(A, x)$. Which of these strings are in $L(M_0)$? (We will say that such strings are **accepted by** M_0.)

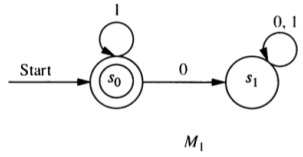
   a. λ
   b. 0
   c. 1
   d. 010
   e. 1011

Here are three automata:

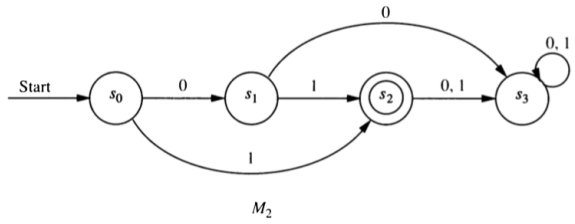
4. For each of the machines M_1, M_2, M_3, compute

   a. $f^*(s_0, 10)$
   b. $f^*(s_0, 1011)$
   c. $f^*(s_1, 1011)$

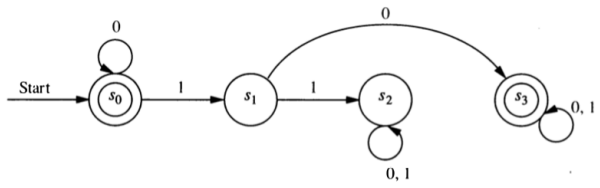
5. For each of the machines M_1, M_2, M_3, determine the language recognized.



M_1



M_2



M_3

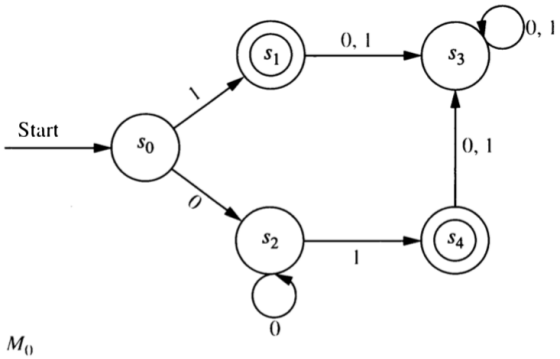
6. Create automata that recognize each of the following languages.

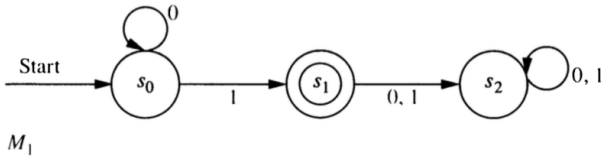
   a. The set of bitstrings that begin with two 0's
   b. The set of bitstrings that contain exactly two 0's
   c. The set of bitstrings that contain at least two 0's
   d. The set of bitstrings that contain two consecutive 0's (anywhere in the string)
   e. The set of bitstrings that do not contain two consecutive 0's anywhere
   f. The set of bitstrings that end with two 0's

3.4 Equivalent Automata and Grammars

We will say that two automata M and N are equivalent if $L(M) = L(N)$, that is, if they recognize the same language. Similarly, two grammars are equivalent if they generate the same language.

7. What must you do to show that two automata are **not** equivalent?
8. What must you do to show that two automata **are** equivalent.
9. Are the two automata represented below equivalent? Explain.





3.5 Nondeterministic Automata

The finite state automata we have seen so far are often called **deterministic finite-state automata** or DFAs. There is a generalization called a **non-deterministic finite-state automaton** or NFA. The only difference is how the transition function is specified. For an NFA, the transition function has the form

$$f : S \times I^* \to P(S)$$

where S is the set of states, I is the input alphabet, and $P(S)$ is the powerset of S. (The powerset of S is the set of all subsets of S. So the transition function for an NFA returns a *set* of states.)

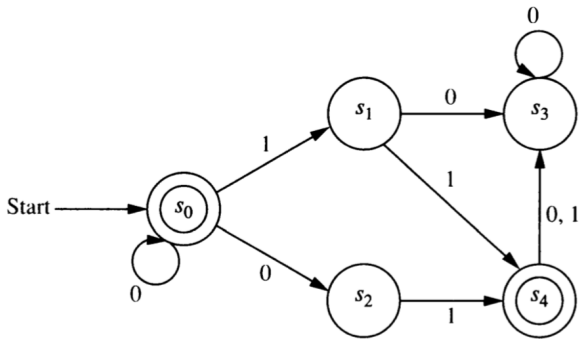
Here is a tabular description of the transition function for an NFA N_0 with start state A and final states C and D.

N_0								
state	A	A	B	B	C	C	D	D
letter	0	1	0	1	0	1	0	1
f	$\{A, B\}$	$\{D\}$	$\{A\}$	$\{B, D\}$	$\{\}$	$\{A, C\}$	$\{A, B, C\}$	$\{B\}$

10. Draw the graph representation. How will the graph of an NFA differ from the graph for a DFA?
11. How do we define the language recognized by an NFA? [Check your answer before proceeding.]
12. Which of these strings are accepted by N_0?

a. 01
b. 011
c. 00
d. 0000
e. λ
f. 010101

Here is the graph of an NFA N_1.



13. Make a table showing the transition function for N_1.
14. Which of the following strings are accepted by N_1?

    a. λ
    b. 00
    c. 01
    d. 010
    e. 011
    f. 001

15. What is $L(N_1)$?
16. Create a DFA that recognizes $L(N_1)$.
17. Given an NFA N, is it always possible to create a DFA M that recognizes the same language? If so, explain how. If not, provide an example N and explain why $L(N)$ cannot be recognized by a DFA.

4 Regular Expressions and Languages

4.1 Operations on Languages

Since languages are sets, we can use set operations like $\cup$ and $\cap$ to create new languages. But there are some additional operations that we will use.

- concatenation of strings: xy is the concetanation of strings x and y (x followed by y)
- concatenation of sets: $AB = \{xy \mid x \in A \wedge y \in B\}$
- power: $A^0 = \emptyset$; $A^1 = A$; $A^{n+1} = A^n A$. (Power is iterated concatenation.)
- Kleene star: $A^* = \cup_{n=0}^{\infty} A^n$

1. Which of these strings belong to $\{0, 01\}^*$?
   a. 01001
   b. 10110
   c. 00010
2. Which of these strings belong to $\{101, 111, 11\}^*$?
   a. 1010111
   b. 1011011
   c. 1110111
   d. 11110111

4.2 Regular Expressions

4.2.1 Syntax

A regular expression over an input alphabet I is defined recursively.

- $\boldsymbol{\emptyset}$ and $\boldsymbol{\lambda}$ and regular expressions.
- $\boldsymbol{x}$ is a regular expression for each $i \in I$.
- If $\boldsymbol{A}$ and $\boldsymbol{B}$ are regular expressions, then
  - $\boldsymbol{A}^*$ is a regular expression
  - $(\boldsymbol{A} \cup \boldsymbol{B})$ is a regular expression
  - $(\boldsymbol{AB})$ is a regular expression

4.2.2 Semantics

Each regular expression represents a language as follows.

- $\boldsymbol{\emptyset}$ represents the empty langauge: $\emptyset$
- $\boldsymbol{\lambda}$ represents language that contains only the empty string: $\{\lambda\}$
- For each $x \in I$, $\boldsymbol{x}$ represents the language containing only x: $\{x\}$
- If $\boldsymbol{A}$ and $\boldsymbol{B}$ are regular expressions representing languages A and B, then
  - $(\boldsymbol{AB})$ represents AB
  - $(\boldsymbol{A} \cup \boldsymbol{B})$ represents $A \cup B$ [Note: This is often written `(A|B)` in programming languages.]
  - $\boldsymbol{A}^*$ represents A^* [Note: This is often written as `A*` in programming languages.]

The langauges represented by regular expressions are called **regular languages** (how creative).

3. In Rosen 7, the definition of regular expression is missing boldface in a couple places. Find them.

> The *regular expressions* over a set I are defined recursively by:
>
> the symbol $\emptyset$ is a regular expression;
> the symbol λ is a regular expression;
> the symbol $\boldsymbol{x}$ is a regular expression whenever $x \in I$;
> the symbols $(\mathbf{AB})$, $(\mathbf{A} \cup \mathbf{B})$, and $\mathbf{A}^*$ are regular expressions whenever $\mathbf{A}$ and $\mathbf{B}$ are regular expressions.

4. What is the difference between $\mathbf{1}$ and 1? between $\boldsymbol{\lambda}$ and λ? between $\boldsymbol{\emptyset}$ and $\emptyset$?
5. Describe in words the languages represented by each of these regular expressions:

   a. $\mathbf{10^*}$ b. $\mathbf{(10)^*}$ c. $\mathbf{0 \cup 01}$ d. $\mathbf{0^* \cup 01}$ e. $\mathbf{0(0 \cup 1)^*}$ f. $\mathbf{(0 \cup 01)^*}$
6. Can each of the languages above be recongized by an NFA? a DFA?
7. What would we need to do to show that every regualar language can be recognized by an NFA? Give it a try and see how far you get. Which parts are easy? Which are trickier? Why?

4.3 Recognizing Regular Languages with NFAs

1. Show that the empty language ($\emptyset$) is recognized by an NFA.
2. Show that the language $\{\lambda\}$ is recognized by an NFA.
3. Show that if $a \in I$, then the langauge $\{a\}$ is recognized by an NFA.
4. Show that if A and B are each recognized by an NFA, then AB is, too.

   [Hint: Let N_A and N_B be the NFAs that recognize A and B. How can you use them to build a new NFA that recognizes AB?]
5. Show that if A and B are each recognized by an NFA, then $A \cup B$ is, too.

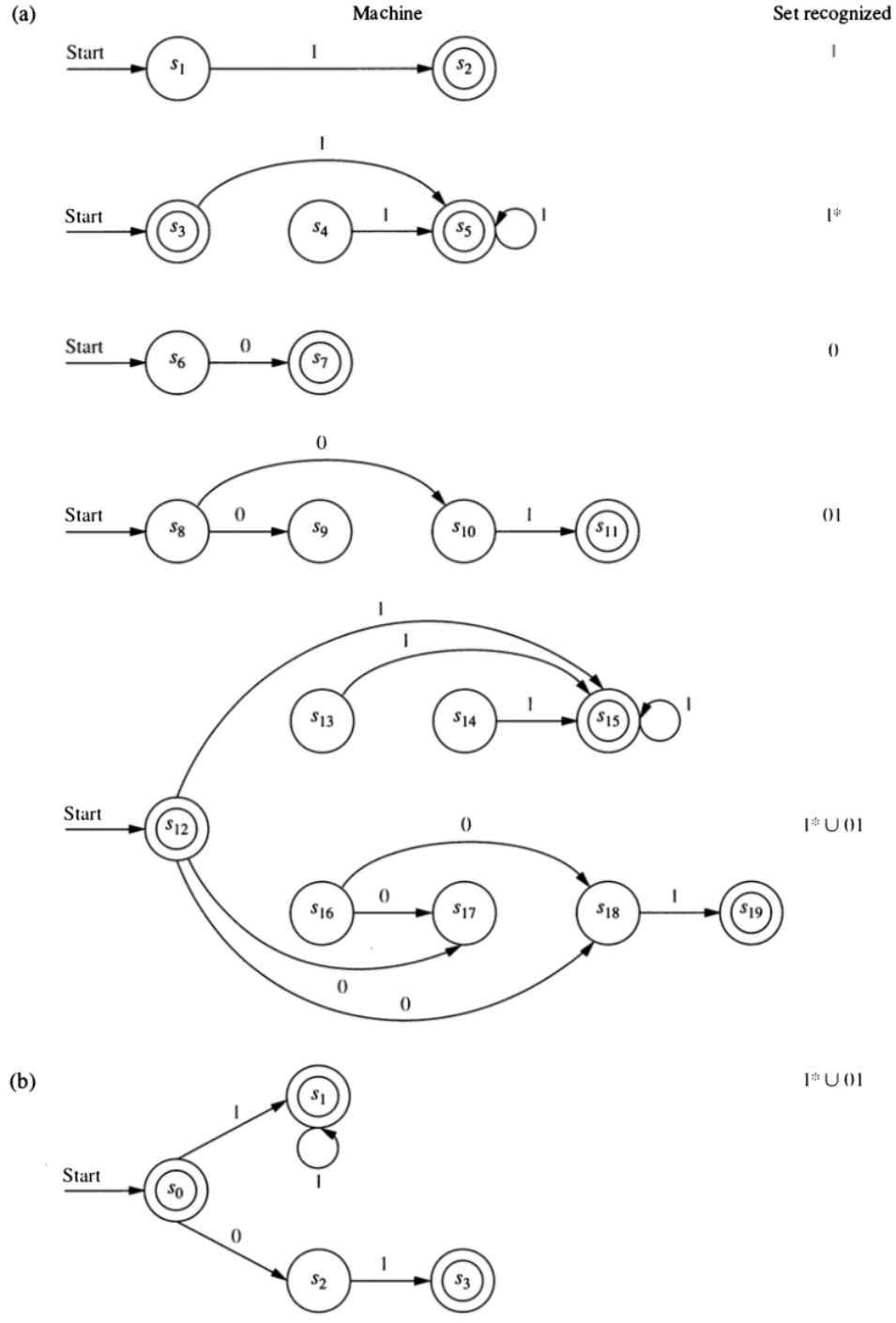
   [Hint: Let N_A and N_B be the NFAs that recognize A and B. How can you use them to build a new NFA that recognizes $A \cup B$?]
6. Show that if A is recognized by an NFA, then A^* is, too.

   [Hint: Let N_A be the NFA that recognizes A How can you use it to build a new NFA that recognizes $A \cup B$?]
7. Explain how 1–6 above show that every regular language is recognized by an NFA.
8. Explain how 7 implies that every regular language is recognized by a DFA.
9. The method just outlined is automatic (you could fairly easily write a computer program to do the translation from regular expression to NFA), and it provides a proof that all regular languages can be recognized by an NFA. But it might produce an NFA with more states than is minimally required. Create an NFA that recognizes $\mathbf{1}^* \cup \mathbf{01}$. Do this two ways:

   a. Following the steps outlined in 1–6.
   b. Any way you like, but using fewer states than in part a.
   c. What is the smallest (fewest states) NFA you can find that recognizes $\mathbf{1}^* \cup \mathbf{01}$?
10. Show that if A is recognized by an NFA, then the complement of A ($\overline{A} = A^C$) is too.

    [Hint: There is an easy way and a hard way – use the easy way.]

4.4 Solution to problem 4.3.9



5 Regular Grammars and Automata

5.1 Grammars to Automata

1. Construct a finite state automaton that recognizes the langauge generated by the following regular grammar.

   - terminals: 0 and 1
   - nonterminals: S and A
   - start symbol: S.
   - production rules: $S \to 1A$, $S \to 0$, $S \to \lambda$, $A \to 0A$, $A \to 1A$, $A \to 1$

2. Construct a finite state automaton that recognizes the langauge generated by the following regular grammar.

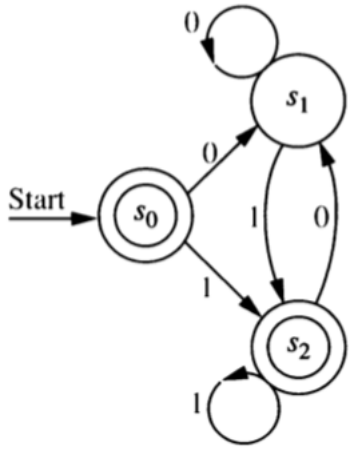
   - terminals: 0 and 1
   - nonterminals: S, A, B
   - start symbol: S.
   - production rules: $S \to 1A$, $S \to 0$, $A \to 0B$, $A \to 1A$, $A \to 1$ $B \to 0A$, $B \to 1B$, $B \to 0$

3. Explain how *any* regular grammar G can be converted into an NFA N such that $L(N) = L(G)$.

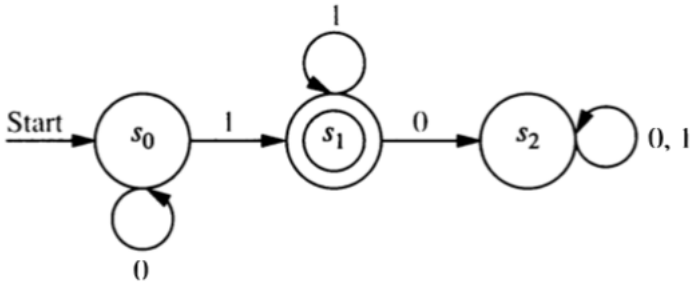
   a. What is a regular grammar? (What sorts of production rules are allowed?)
   b. What will the states of N be? How many will there be?
   c. What state will be the start state?
   d. Which states will be final (ie. accepting) states?
   e. How does each rule in G get converted into a part of N? [Go through each kind of rule a regular grammar may have.]

5.2 Automata to Grammars

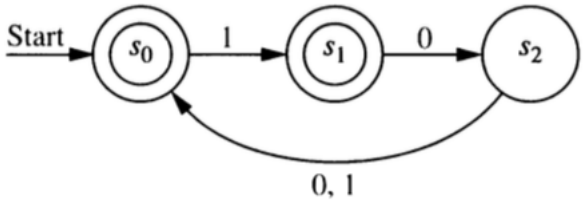
4. Show that every DFA or NFA is equivalent to a DFA or NFA with a "lonely start state". A lonely start state is a start state that can never be returned to. (In terms of the graph, its in-degree is 0.)
5. Convert each of the automata in the next three problems into an automoton with a lonely start state.
6. Construct a regular grammar that generates the language accepted by this finite state automaton.



7. Construct a regular grammar that generates the language accepted by this finite state automaton.



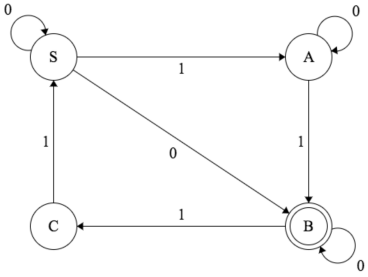
8. Construct a regular grammar that generates the language accepted by this finite state automaton.

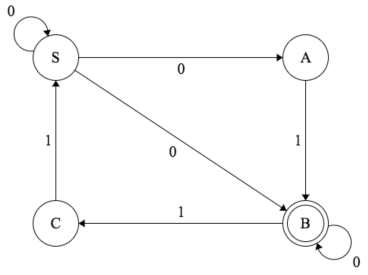


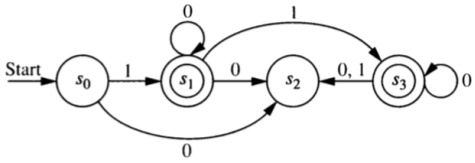
9. In this problem you will show that any DFA or NFA with a lonely start state is equivalent to a regular grammar by showing how to construct a grammar G from such an automaton.
   a. What will the terminals and nonterminals in your grammar be?
   b. What will the start symbol be?
   c. How are the production rules created?
   d. Why was it necessary to have a lonely start state?

5.3 Some Review

1. Convert each of the following NFA's into an equivalent DFA. (Rembember: equivalent means that they recognize the same language.) Do this using our general algorithm for this task. For the first two, the start state is S.







6 Regular Langauges

6.1 Equivalent Definitions

We have been working toward a result that says the following are all equivalent.

1. L can be described by a regular expression.
2. L is recognized by a DFA.
3. L is recognized by an NFA.
4. L is generated by a regular grammar.

The proof that these are equivalent amounts to describing algorithms that can convert from one representation to another. In particular we have shown

a. $1 \Rightarrow 3$
b. $2 \Rightarrow 3$
c. $3 \Rightarrow 2$
d. $2 \Rightarrow 4$
e. $4 \Rightarrow 3$

Still missing from our list is

f. $2 \Rightarrow 1$

Once we have that, we will see that all 4 definitions are equivalent.

Your Mission (You must choose to accept it)

For each conversion a – e, write a brief description of the algorithm. Your description should include the most important "big idea" and also any potential "gotchas", things you have to be careful about or might forget about.

Bonus: Can you figure out how to do conversion f?

6.2 A language that is not regular

Since we now have several equivalent ways to show that a langauge is regular, we also have several ways to show that a languages is not regular.

Here is a language that is not regular: $L = \{0^n 1^n \mid n = 0, 1, 2, ...\}$.

1. What are our options for showing the language is not regular?
2. Which do you think will be easiest?
3. See if you can give a convincing argument that L is not regular.
4. What other languages can you construct that are not regular using the same basic idea idea?

6.3 Solutions

Compare your descriptions to the descriptions below. Did you leave out anything important? Do the solutions below leave out anything important?

1a. regex → NFA:

- 6 parts:
  - 3 base cases – pretty easy
  - 3 that show union, concatenation and Kleene star – these are the "interesting" part
- union: $A \cup B$ where $A = L(N_1)$ and $B = L(N_2)$.
  - keep all states and transitions from N_1 and N_2
  - add one new start state S; S is final if either start state of N_1 or start state of N_2 is final
  - add some additional transitions
    - if $S_1 \xrightarrow{x} A$, add $S \xrightarrow{x} A$
    - if $S_2 \xrightarrow{x} B$, add $S \xrightarrow{x} B$
- concatenation: AB where $A = L(N_1)$ and $B = L(N_2)$.
  - keep all states and transitions from N_1 and N_2
  - start state from N_1 is the start state
  - only final states from N_2 are final
  - add some additional transitions
    - if $A \xrightarrow{x} F$ (a final state in N_1), then add add $A \xrightarrow{x} S_2$ (start state in N_2)
- Kleene star: $A = L(N_1)$
  - keep all states and transitions from N_1
  - add a new start state that is a final state (because $\lambda \in A^*$)
  - add new transitions from new start state (S)
    - if $S_1 \xrightarrow{x} A$, add $S \xrightarrow{x} A$ (includes case where $A = S_1$)
  - add new transitions to old start state (S_1)
    - if $A \xrightarrow{x} F$, where F is final, add $A \xrightarrow{x} S_1$
    - If we are at the end of a string in $L(N_1)$, this lets us "restart"

1b. DFA → NFA

- easy: a DFA "is" an NFA

1c. NFA → DFA

- DFA states are **subsets of NFA states** (think: magnets on white board)
- A is final in DFA if A contains a final state of NFA
- Remember that {} may be needed as one of the states in the DFA

1d. DFA (or NFA) → reg grammar

- $A \xrightarrow{x} B$ becomes
  - $A \to xB$
  - also $A \to x$ if B is final state
- If S is final in NFA/DFA, then add $S \to \lambda$

- Number of rules = number of transitions + number of transitions to final states + 1 more if the start state is final

1e. reg grammar $\to$ NFA

- states: non-terminals of grammar + one additional state F (a final state)
  - Start symbol is also start state
  - If $S \to \lambda$ is a rule, then S is a final state
- production rules become transitions in NFA
  - $A \to xB$ becomes $A \xrightarrow{x} B$
  - $A \to x$ becomes $A \xrightarrow{x} F$ (the added final state)

7 DFA/NFA to Regular Expression

7.1 GNFA (Generalied NFA)

A GNFA (Generalized NFA) is like an NFA but the edges may be labeled with any regular expression. One way of obtaining a regular expression from a DFA or NFA uses an algorithm that works with GNFAs.

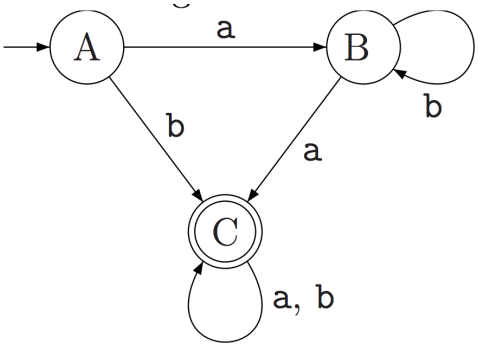
7.2 Algorithm for converting DFA/NFA to Regular Expression

Suppose we want to find an equivalent regular expression for some DFA or NFA. Here is an algorithm to do so.

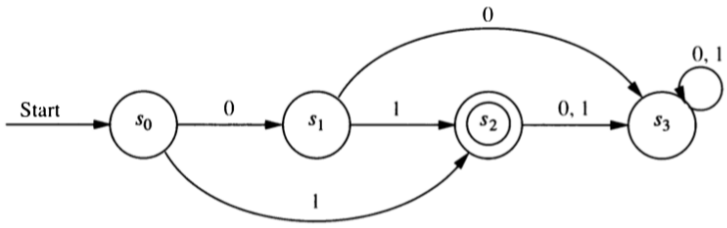
1. Modify the the original machine.
   a. Add a **new start state** with a λ transition to the old start state. (We can skip this step if the start state is already lonely (has in-degree 0).)
   b. Add a **new final state** with a λ transition from all old final states. (We can skip this step if there is exactly one final state and it has out-degree 0.)
   c. The new final state will be the **only final state**.
   d. **Replace mutliple edges** between any pair of states A and B with one edge labeled with the union of the labels of the original edges. [Example $A \overset{0,1}{\to} B$ becomes $A \overset{0\cup 1}{\to} B$.]
2. Pick an internal state (not the start state or the final state) to "rip out". Let's call that state R.
   a. If R doesn't have a self-loop, replacejevery $A \overset{r_{\text{in}}}{\to} R \overset{r_{\text{out}}}{\to} B$ with $A \overset{(r_{\text{in}} r_{\text{out}})}{\longrightarrow} B$.
   b. If R has a self-loop labeled r_{self}, replace every $A \overset{r_{\text{in}}}{\to} R \overset{r_{\text{out}}}{\to} B$ with $A \overset{(r_{\text{in}} r^*_{\text{self}} r_{\text{out}})}{\longrightarrow} B$.
   c. If this results in multiple edges between two states A and B, replace them with one edge labeled with the union of their labels.
3. Repeat step 2 until the only states left are the start state and the final state. There will be only one transition remaining, and its label will be a regular expression for the language recognized by the original DFA or NFA.

7.3 Examples

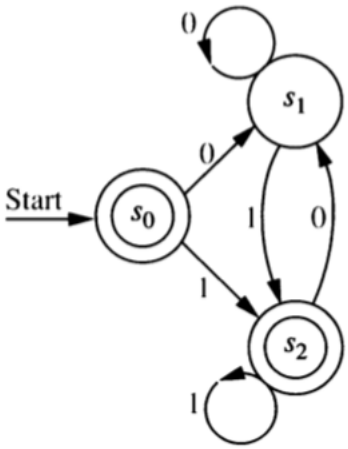
1. Systematically convert this DFA into an equivalent regular expression.



2. Systematically convert this DFA into an equivalent regular expression.



3. Systematically convert this DFA into an equivalent regular expression.



You can find additional worked examples online at places like

- https://courses.cs.washington.edu/courses/cse311/14sp/kleene.pdf
- https://courses.engr.illinois.edu/cs374/sp2019/extra_notes/01_nfa_to_reg.pdf

But note that sometimes the notation isn't quite the same.

- ε is sometimes used for what we have called λ.
- $+$ is sometimes used in place of $\cup$.

7.4 Review problems

1. Show that if A is a regular language, then $\overline{A}$ is also a regular language.
2. Show that if A and B are regular languages, then $A \cap B$ is also a regular language.
3. Convert the machines on this sheet to equivalent regular grammars.

8 Regular Expressions in Python

8.1 Getting your python environment working

To experiment with regular expressions in python, you will need access to python 3 and the `requests` module. If you don't already have python running on your machine, here are some options.

8.1.1 repl.it

This python-in-a-browser seems able to do what we need (based on minimal testing)

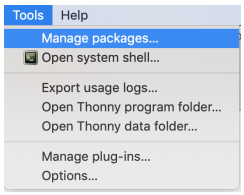
https://repl.it/languages/python3

8.1.2 Thonny

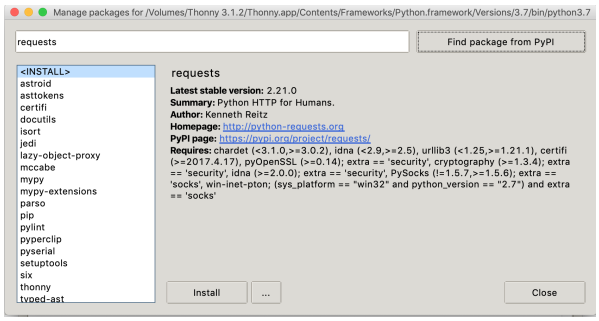
If you prefer to install a simple versions of python locally, try Thonny.

1. Download and install Thonny from https://thonny.org/.
2. Make sure the `requests` module is available.

   In Thonny, you can install packages from the **tools > manage packages** menu:



   Then search PyPI for `requests` and hit install.



8.1.3 Testing your python setup

If you can run the following and get the same output, you should be OK.

```
import re
import requests
# Download St. Augustine's confessions from CCEL and print it
url = 'https://www.ccel.org/ccel/augustine/confess.txt'
r = requests.get(url)
book = r.text
print(len(book))
```

```
## 708873
```

```
print(book[1015:1120])
```

```
##         Everywhere God wholly filleth all things, but neither heaven nor
##         Earth containeth him.
##
```

```
foo = re.compile("aug")
print(foo.search(book))
```

```
## <_sre.SRE_Match object; span=(4314, 4317), match='aug'>
```

9 Turing Machines

The Turing machine model pre-dates electronic computers and comes from the work of Hilbert (early 1900's) and Turing and Church (1930)'s.

- Not trying to model (electronic) computers, but computation or algorithm
- Identified three essential ingredients: _____, _____, _____
  - implicitly there is a fourth ingredient: _____
- The Turing machine is a specific formalization of these basic ingredients

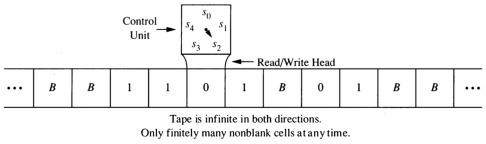
9.1 Definition of a 1-tape Turing Machine (Syntax)

A Turing machine consists of the following:

- **input alphabet**: A set of symbols used for inputs.
- **tape alphabet**: Input alphabet plus and at least one additional symbol, the **blank symbol**.
- **states**: A finite set.
- **initial/start state**: one of the states to start in
- **final states**: some of the states are designated as final or accepting states. (In some variations there are two kinds of final states: accepting and rejecting.)
- **program**: A list of **instructions**. Each instruction has 5 parts:
  - `<current tape symbol>`: any tape symbol
  - `<current state>`: any state
  - `<new tape symbol>`: any tape symbol
  - `<new state>`: any state
  - `<direction>`: left, right, or stay

9.2 Execution of a Turing Machine program (Semantics)

Imagine a machine with a 2-way infinite tape divided into cells. Each cell contains exactly one symbol from the tape alphabet. At the start of the execution, the input is written on the tape, the read/write head is located at the left most symbol of the input, and all cells that don't contain part of the input contain the blank symbol.



At each step in the execution of a Turing machine program, the read/write head is located at one cell of the tape. The program uses the current state and contents of the tape at that position to determine its action:

- write a symbol in the current cell (overwriting what was there before),
- move the head left or right (or stay put)
- update the state

This is repeated until one of two things happens:

- there is no instruction to execute, or
- the state is a final state

When either of these two things happens, the machine **halts**.

9.2.1 ASCII Representations of Turing Machines

There are several websites that provide Turing machine simulators. Here are a few:

- http://www.6by9.net/z/projects/turingMachine/turingMachineApplet/ – My favorite, but no longer usable since it was written in java by Calvin student Eliot Eschelman back when java applets were still a thing.
- http://morphett.info/turing/turing.html – plain output, and less flexible (initial state must always be called 0, for example), but the program encoding is terse.
- https://turingmachinesimulator.com/ – nice looking and more flexible, but each instruction takes up 3 lines (current state and tape symbol on one line; new state, direction, and new symbol on the next line; then a blank line)

Each uses a slightly different ASCII representation of a Turing Machine. They are all pretty similar but differ in

- How the input alphabet, tape alphabet, start state and accepting states are described
- How they represent the blank tape symbol. (Often there is a fixed symbol for this like `b`, `B` or `_`.)
- Whether they are case sensitive, disallow certain characters, observe white space, etc.
- How left/right/stay are denoted
- The order in which the 5 parts of an instruction are given
- Which of various variations on the the 1-tape Turing machine theme they support (and how you tell the simulator which flavor

9.3 Examples

9.3.1 Example 1

```
; in style of <http://morphett.info/turing/turing.html>
; state symbol symbol direction state
; s0 is initial state
0  * * * s0
s0 0 0 r s0
s0 1 1 r s1
s0 _ _ r halt
s1 0 0 r s0
s1 1 1 l s2
s1 _ _ r halt
s2 1 0 r halt
```

1. For each input string below, determine the configuration (tape contents, location of read-write head, and state) of the Turing machine when it halts.

   a. λ
   b. 01
   c. 0101
   d. 010110
   e. 001

9.3.2 Example 2

```
; [init] is initial state
0 * * * [init]
[init] 0 0 r [10]
[init] 1 0 r [01]
[init] _ _ * halt-init
[00] 0 0 r [10]
[00] 1 1 r [01]
[00] _ _ * halt-00
[01] 0 0 r [11]
[01] 1 1 r [00]
[01] _ _ r [01]
[10] 0 0 r [00]
[10] 1 1 r [11]
[11] 0 0 r [01]
[11] 1 1 r [10]
```

2. For each input string below, determine the configuration (tape contents, location of read-write head, and state) of the Turing machine when it halts.

   a. λ
   b. 0110
   c. 01111
   d. 01110

9.3.3 The language recognized by a Turing Machine

We will say that a Turing machine M accepts a string x (by state) if on input x it halts in a final state. The language recognized by a Turing machine is

$$L(M) = \{x \mid M \text{ accepts } x\}$$

3. What languages do the previous two machines recognize?

9.3.4 Designing a Turing Machine

4. Design a Turing machine that recognizes $(\mathbf{0} \cup \mathbf{1})\mathbf{1}(\mathbf{0} \cup \mathbf{1})^*$
5. Design a Turing machine that recognizes $\{0^n 1^n \mid n \geq 0\}$. This shows that Turing machines can recognize languages that are not regular.

Turing machines can also compute functions. The output of a Turing machine is the tape contents when it halts.

6. Design a Turing machine that adds in unary. Use the input alphabet $\{1, +\}$.

   In unary, a non-negative integer n is represented by $n + 1$ 1's. So 3 is represented as 1111.

   Example: Our machine should convert `111+1111` into _____.

A Turing machine computes **neatly** if when it halts (a) the read/write head is on the left-most non-blank tape symbol (or any blank symbol if the tape is all blanks), and (b) there are no internal blanks (ie, all the non-blank symbols are contiguous on the tape).

7. Modify your previous machine so that it computes neatly.

9.4 What Can Turing Machines Do?

9.4.1 Turing Computable Languages

A language L is **Turing computable** if it is recognized by some Turing machine **that always halts**, that is,

- $L = L(M)$ for some Turing machine M.
- M halts on every input. (If $x \in L$ it halts in an accepting state.)

1. Show that every regular language is Turing computable.

   [Hint: What must you do to demonstrate this?]

9.4.2 The Halting Problem

The **Halting Problem** is a famous example of a language that is not Turing computable. By the **Church-Turing Thesis**, this would mean it is not computable by any means.

Why is it called the Halting *Problem*, you ask, instead of the Halting *Language*? This goes back to a description of languages as answers to yes/no questions. Each such problem (ie, language) an be described by an **instance** and a **question**. Here are some examples:

Euler

- Instance: A Graph G
- Question: Does G have an Euler Circuit?

3-Colorability

- Instance: A Graph G
- Question: Is G 3-colorable? (Can we color the vertices with 3 colors so that no adjacent vertices are the same color.)

SAT

- Instance: A first order logical formula φ (made with $\wedge$, $\vee$, $\neg$ and some logical variables).
- Question: Is φ satisfiable? (Can we set the variables to TRUE/FALSE in a way that makes the entire formula true?)

Implicit in each of these descriptions is some encoding scheme by which the objects mentioned in the instance are coded as bit strings. Typically, if we are only interested in whether a problem is Turing computable or not, it does not matter which coding scheme is used as long as it is systematic.[1]

An entire book [Garey and Johnson, 1979] of such problems (and whether or not they were NP-complete) was produced in the 1970's by Garey and Johnson and popularized this description. So you will see languages referred to as problems throughout the CS literature.

Here is a description of the Halting Problem.

- Instance: A Turing Machine M and an input string x.
- Question: Does M stop on input x? (This is written $M(x)\downarrow$.)

2. Show that that Halting Problem is not Turing computable.

[1] The coding scheme can affect the efficiency of a Turing machine computation. Generally it is required that the encoding be reasonably efficient and not bloated or wasteful of space.

9.4.3 Variations on the Turing Machine

There are many variations on the Turing machine. Here are some examples.

- Instead of having a single tape, a Turing machine can have multiple tapes.
- Instead of being 2-way infinite, the tape(s) can have a left end. These are called 1-way infinite tapes.
- Turing machines may be required to move the head at each step (so "stay" is not an option).
- Multi-track tapes aren't really a variation, but a handy way of thinking about a tape alphabet. (And some simulators makes this easy to do.) Since the tape alphabet may be anything, if we want to simulate writing two symbols from A, we can use a tape alphabet of $A \times A$ or even $A \cup (A \times A)$. One element of $A \times A$ represents two elements from A (one on the "top track" and one on the "bottom track").
- Instead of having a 1-dimensional tape, a tape can be a 2-dimensional grid of cells. The read/write head can move up and down in addition to left and right.

Each of these variations is equivalent to the standard Turing machine with one 2-way infinite tape. So are many other models of computation. This has led to the so-called **Church-Turing Thesis**. Here is one statement of that thesis:

> A function on the natural numbers is computable by a human being following an algorithm, ignoring resource limitations, if and only if it is computable by a Turing machine.

3. Give a high level description of how to convert a Turing machine with 2-way infinite tape into a Turing machine with a 1-way infinite tape.
4. Give a high level description of how to convert a 2-tape machine into a 1-tape machine.
5. **Nested Parens** Write a Turing machine recognizes all strings that use the input alphabet $\{(,), *\}$ and the parentheses are properly nested. (The asterisks may be anywhere.)
   a. First try this with a two-tape machine.
   b. Then try this with one two-track tape.
   c. For an extra challenge, try doing it with a one-tape machine with tape alphabet $\{(,), *, \#\}$

9.5 Running time of a Turing machine

The running time of a Turing machine on input x is the number of steps before it halts. The conversions above typically change the running time, but machines that run in polynomial time are converted into machines that run in polynomial time (typically with a different degree polynomial).

9.5.1 P and NP

This provides a formal definition of the famous set P. P is the set of all languages that can be recognized by a Turing machine in polynomial time, that is, the number of steps the machine takes on an input of length n is bounded by $p(n)$ for some polynomial p.

A **non-deterministic Turing machine** can have multiple transitions from the same state/tape contents configuration. A non-deterministic TM accepts an input x if there is a way of choosing transitions that causes the machine to halt in a final state. (This is similar to how non-deterministic automata were defined). NP is the set of all languages that can be recognized by a non-deterministic Turing machine in polynomial time.

The "P vs. NP question" is whether P and NP are the same or different. To date, this question is unresolved. [2]

[2] Solving this could make you both rich and famous.

References

M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979.