Lecture 5 - Nonregular Languages

Eric A. Autry

Last time: Converting DFAs to Regular Expressions

This time: Nonregular Languages

Next time: Infinities

Homework 2 on Sakai and due **this THURSDAY the 20th.**

Midterm **next THURSDAY the 27th!** If you have scheduling conflicts, please email me today.

Even vs Odd Positions

For this automata theory section of the course, we will index strings starting at 1.

Consider the string ‘abcde’:

a	b	c	d	e
1	2	3	4	5

Even vs Odd Positions

What about the empty string? It doesn't have any odd positions.

The following statements are **equivalent**:

- Every odd position is a 1.
- There does **not** exist an odd position that is **not** a 1.

The empty string has no odd positions, so the first statement is confusing.

However, the second statement is clearly true for the empty string, and therefore **both** statements are true.

Lexer/Tokenizer

Nested comments **are not allowed** in C.

The problem is that it introduces ambiguity: different lexers will result in different behaviors.

What should we do with:

$$/ * */ * /$$

Well, if we're greedy, this seems to work fine.

Lexer/Tokenizer

But if we're greedy, what happens here?

```
/* Here is some code: */
code
.
.
.
code
/* No more code. */
```

In order to be practical, a "traditional" comment in C starts with the characters /* and runs until the next occurrence of */.

Nested comments are forbidden.

IMPORTANT CORRECTION

While it looks like a nested comment, the following string is a valid C comment (it actually compiles despite the fact that some editors, like `vim`, display warnings):

$$/*/**/$$

A “traditional” comment in C starts with the characters $/*$ and runs until the next occurrence of $*/$.

So, for this string, the second occurrence of $/*$ is ignored.

Lexer/Tokenizer

SUMMARY:

The following nested comment will **reject**:

```
/* comment /* nested! */ uh oh */
```

The following nested comment will **reject**:

```
/* comment */ no longer a comment */
```

The following comment will compile and **should be accepted**:

```
/* comment /* tricking the prof! */
```

Regular Languages

Question: are all languages regular?

Ex:

$$A = \{w \mid w \text{ has the same number of } 1\text{s and } 0\text{s}\}.$$

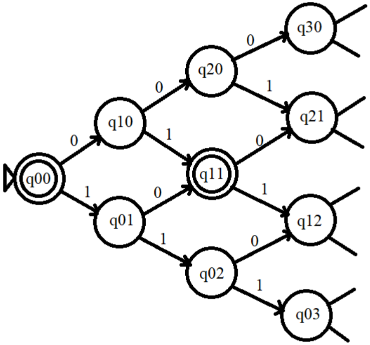
How can we build a machine to recognize this language?

Regular Languages

$$A = \{w \mid w \text{ has the same number of 1s and 0s}\}.$$

Attempt 1: Let's try to keep track of both the number of 0s and the number of 1s.

Define the state $q_{ij} = i$ number of 0s and j number of 1s.



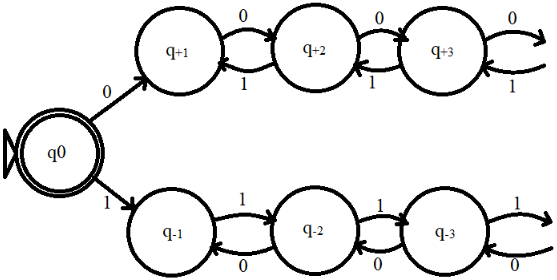
Infinite number of states! No longer a **finite** state machine...

Regular Languages

$$A = \{w \mid w \text{ has the same number of 1s and 0s}\}.$$

Attempt 2: Let's try to keep track of how many more 0s than 1s.

Define the state $q_{+i} = i$ more 0s than 1s, and the state $q_{-i} = i$ fewer 0s than 1s.



Still an **infinite** number of states!

(Note: same number of states as the first attempt)

Nonregular Languages

Some languages are not regular languages and cannot be recognized by any DFAs, NFAs, or Regular Expressions.

Or rather:

MOST languages are not regular languages and cannot be recognized by any DFAs, NFAs, or Regular Expressions.

Nonregular Languages

Can we prove if a language is regular or not?

Yes.

- To prove that it is, build a DFA, NFA, or Regular Expression that recognizes it.
- To prove that it is not, use the Pumping Lemma...
  - We will not cover this lemma in this overview of automata theory.

Context Free Grammars

Context Free Grammars (CFGs) are a more powerful method of describing languages than we have seen before.

They were first used in the study of human languages to help understand sentence structures.

An important application of CFGs occurs during the compilation of programming languages: most compilers contain a component called a **parser** that extracts the meaning of a program prior to the generation of compiled code.

Context Free Grammars

CFGs are defined recursively with a list of **substitution rules.**

Ex: (alphabet: $\{0, 1\}$)

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \varepsilon \end{aligned}$$

Start with a single S, then make any number of substitutions until you have eliminated any symbols that are not in the alphabet.

So we could generate the string 000111 through the following:

$$S \rightarrow 0S1 \rightarrow 00S11 \rightarrow 000S111 \rightarrow 000\varepsilon111 = 000111$$

This is the language: $\{0^n1^n \mid n \geq 0\}$

Context Free Grammars

What if we wanted to describe the language:

$$\{w \mid w \text{ has the same number of } 1\text{s and } 0\text{s}\}?$$

$$\begin{aligned} S &\to S01 \\ S &\to S10 \end{aligned} \qquad \begin{aligned} S &\to 0S1 \\ S &\to 1S0 \\ S &\to \varepsilon \end{aligned} \qquad \begin{aligned} S &\to 01S \\ S &\to 10S \end{aligned}$$

Ex: we can now generate 101100:

$$S \to (1S0) \to 1(S10)0 \to 1(0S1)100 \to 10\varepsilon1100 = 101100$$

OR:

$$S \to (1S0) \to 1(01S)0 \to 101(1S0)0 \to 1011\varepsilon00 = 101100$$

Ambiguity can be resolved by using *Chomsky normal form*.

Pushdown Automata

Just like regular languages, these new context free languages have machines that can recognize them: Pushdown Automata (PDAs).

These new, more powerful machines, are created by taking an NFA and adding a stack.

- A stack can store infinite memory, but can only be accessed in a specific order:
  First in, last out.

As it turns out, PDAs are equivalent to CFGs. This makes them a useful way to represent parsing.

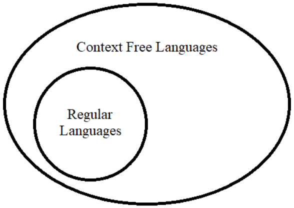
Context Free Languages

What is a PDA that doesn't use its stack?

It is an NFA!

All regular languages are context free, but not all context free languages are regular.

Therefore regular languages are a proper subset of context free languages.



More Pumping

Are all languages context free?

No.

Can we prove whether a language is context free or not?

Yes. We would use what is called the pumping lemma for CFGs. This is material that we will not cover in this class.

Ex:

$$\{w \mid w \text{ is a palindrome}\}.$$

otto *race car* *a man a plan a canal panama*

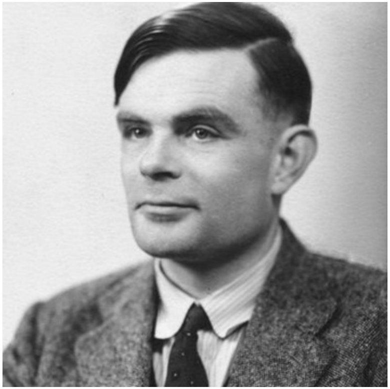
General Purpose Computers

So far we have seen: DFAs, NFAs, and PDAs.

Each has been limited in their capabilities.

So they are too restrictive to use as models for general purpose computers.

Turing Machines



Alan Turing proposed a much more powerful model for computation in 1936.

We call this model the **Turing Machine.**

Turing Machines

A Turing Machine is a finite state machine that is accompanied by an **infinite tape**.

The machine has a read/write head and can:

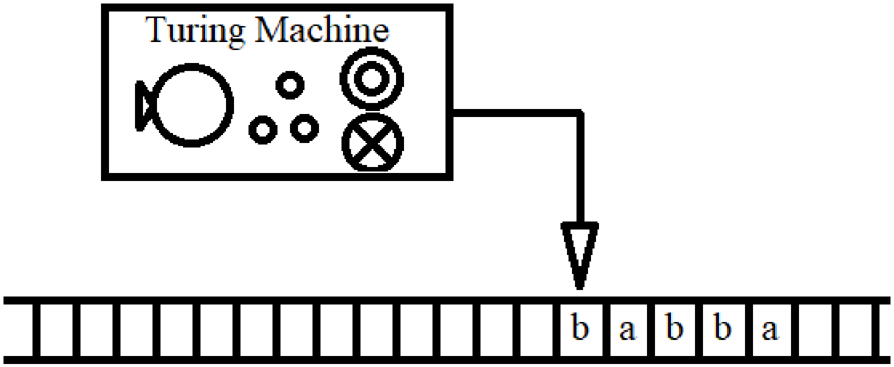
- Read symbols from the tape.
- Write symbols to the tape.
- Move left or right along the tape.

It has a single accept state and single reject state.

- If the machine reaches one of these states, it will immediately halt and either accept or reject the input.
- If the machine never reaches one of these states, it will never halt.

The input to the machine is originally stored on the tape, with the read/write head placed at the beginning of the input string.

Turing Machines



Turing Machines

Ex:

$$\{w \mid w \text{ is a palindrome}\}.$$

We will use the alphabet $\{a, b\}$ for this example.

Accepting string examples:

$$\varepsilon, \quad a, \quad b, \quad aba, \quad abbba, \quad abaabbabbaaba$$

Rejecting string examples:

$$ab, \quad ba, \quad aab, \quad bbaa, \quad abbb, \quad babba$$

Turing Machines

For Turing Machines, we typically describe an *algorithm* that the machine will follow for its computations.

Ex:

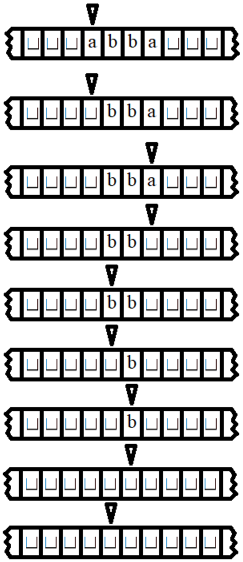
$$\{w \mid w \text{ is a palindrome}\}.$$

Let's work with the test input 'abba'.

1. Read the first letter and cross it off (replace it with an '$\sqcup$').
2. Move to the end of the string that is not yet crossed off.
    - If the letter immediately to the right is already crossed off, halt and accept. This was an odd length string, and so the middle letter doesn't matter.
3. Read the last letter and cross it off (replace it with an '$\sqcup$').
    - If the letters were not the same, halt and reject.
    - Else, keep going.
4. Move to the start of the tape that is not yet crossed off.
    - If the letter immediately to the left is already crossed off, halt and accept. This was an even length string, and we've crossed off all of the matching letters.
5. Go to step 1.

Turing Machines

Let's work with the test input 'abba'. (These are only snapshots of the machine. It can really only move one step at a time.)



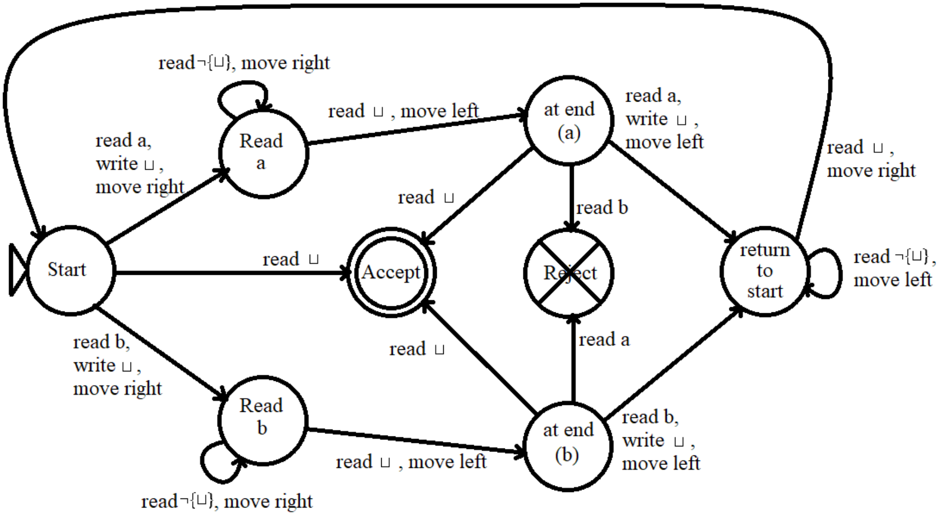
Turing Machines

Let’s build the actual machine...

1. Read the first letter and cross it off (replace it with an ‘$\sqcup$’).
2. Move to the end of the string that is not yet crossed off.
    - If the letter immediately to the right is already crossed off, halt and accept. This was an odd length string, and so the middle letter doesn’t matter.
3. Read the last letter and cross it off (replace it with an ‘$\sqcup$’).
    - If the letters were not the same, halt and reject.
    - Else, keep going.
4. Move to the start of the tape that is not yet crossed off.
    - If the letter immediately to the left is already crossed off, halt and accept. This was an even length string, and we’ve crossed off all of the matching letters.
5. Go to step 1.

Turing Machines

Let’s build the actual machine...



Decidable Languages

If a Turing machine halts on all inputs and either accepts or rejects, the language it recognizes is called a **decidable** language.

When this happens, we say that the Turing machine **decides** the language.