Non-Context-Free Languages, and Intro to Turing Machines

Wednesday, March 10, 2021
Announcements

• **Reminder:** no class next week (Spring Break)
  • 3/15 – 3/19

• HW 5 due tonight
  • 11:59pm EST

• HW 6 released
  • Due Sun 3/28 11:59pm EST (after break)
Flashback: Pumping Lemma for Reg Langs

- The Pumping Lemma describes how strings repeat.

- Strs in a regular lang can (only) repeat using Kleene pattern:
  - Before/during/after parts are independent!

- langs with dependencies are nonregular:
  - E.g., $\{0^n1^n \mid n \geq 0\}$

- Today: How do CFLs repeat?
Repetition and Dependency in CFLs

\[ A \rightarrow 0A1 \]
\[ A \rightarrow B \]
\[ B \rightarrow \# \]

\[ \{0^n#1^n \mid n \geq 0\} \]

Parts before/after repetition are linked

Parts before/after repetition are linked
How Can Strings in CFLs Repeat?

- Strings in regular languages repeat states

- Strings in CFLs repeat subtrees in the parse tree
Pumping lemma for context-free languages  If \( A \) is a context-free language, then there is a number \( p \) (the pumping length) where, if \( s \) is any string in \( A \) of length at least \( p \), then \( s \) may be divided into five pieces \( s = uvxyz \) satisfying the conditions:

1. for each \( i \geq 0 \), \( uv^ixy^iz \in A \),
2. \( |vy| > 0 \), and
3. \( |vxy| \leq p \).

Now there are two pumpable parts. But they must be pumped together!

Pumping lemma  If \( A \) is a regular language, then there is a number \( p \) (the pumping length) where if \( s \) is any string in \( A \) of length at least \( p \), then \( s \) may be divided into three pieces, \( s = xyz \) satisfying the following conditions:

1. for each \( i \geq 0 \), \( xy^iz \in A \),
2. \( |y| > 0 \), and
3. \( |xy| \leq p \).
Non CFL example: \( D = \{ww \mid w \in \{0,1\}^*\} \)

- **Previous:** Showed \( D \) is nonregular w. unpumpable string \( s: 0^p 1 0^p 1 \)
- **Now:** this \( s \) can be pumped according to CFL pumping lemma:

\[
\begin{array}{c}
0^p 1 \\
\{000 \cdots 000\} \quad 0 \quad 1 \\
\{000 \cdots 0001\}
\end{array}
\]

- **CFL Pumping Lemma conditions:**
  - 1. for each \( i \geq 0, \ uv^i xy^i z \in A, \)
  - 2. \( |vy| > 0, \) and
  - 3. \( |vxy| \leq p. \)

This doesn’t prove that the language is a CFL! It only means the counterexample doesn’t work to prove it’s non context-free.
Non CFL example: $D = \{ww \mid w \in \{0,1\}^*\}$

• Choose another string $s$:

If $vyx$ is contained in first or second half, then any pumping will break the match $\times$

0$^p$1$^p$0$^p$1$^p$

So $vyx$ must straddle the middle
But any pumping still breaks the match because order is wrong

• CFL Pumping Lemma conditions:

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$. 
Non CFL example: $D = \{ww \mid w \in \{0,1\}^*\}$

- **Previously:** Showed $D$ is not regular

- **Just Now:** $D$ is not context-free either!
XML Again ...

- We previously said XML sort of looks like the CFL: \( \{0^n1^n | n \geq 0\} \)
  - \text{ELEMENT} \rightarrow \text{<TAG>CONTENT</TAG>}
  - Where \text{TAG} is any string

  But these arbitrary \text{TAG} strings must match!

- So XML also looks like this non-CFL: \( D = \{ww | w \in \{0,1\}^*\} \)

- This means XML is not context-free!
  - \textbf{Note:} HTML is context-free because ...
  - ... there are only a finite number of tags,
  - so they can be embedded into a finite number of rules.

- \textbf{In practice:}
  - XML is parsed as a CFL, with a CFG
  - Then matching tags checked in a 2\textsuperscript{nd} pass with a more powerful machine ...
A More Powerful Machine ...

$M_1$ accepts its input if it is in language: $B = \{w#w \mid w \in \{0,1\}^*\}$

$M_1 = \text{"On input string } w:\$

1. **Zig-zag across the tape** to corresponding positions on either side of the $#$ symbol to check whether these positions contain the same symbol. If they do not, or if no $#$ is found, *reject*. Cross off symbols as they are checked to keep track of which symbols correspond.

Can move to arbitrary memory locations, and read/write to it

Infinite memory, initially starts with input
Turing Machines (TMs)

When it came to eating strips of candy buttons, there were two main strategies. Some kids carefully removed each bead, checking closely for paper residue before eating.

Others tore the candy off haphazardly, swallowing large slabs of paper as they ate.

Then there were the lonely few of us who moved back and forth on the strip, eating rows of beads here and there, pretending we were Turing machines.
Alan Turing

• First to formalize models of computation that we are studying
  • I.e., he invented this course

• Worked as codebreaker during WW2

• Also studied AI
  • Turing Test
Automata vs Turing Machines

• Turing Machines can read and write to “tape”
  • Tape initially contains input string

• The tape is infinite

• Each step: “head” can move left or right

• A Turing Machine can accept/reject at any time

**Definition 3.5**
Call a language *Turing-recognizable* if some Turing machine recognizes it.
Turing Machine Example

$M_1$ accepts inputs in language $B = \{w\#w \mid w \in \{0, 1\}^*\}$

$M_1$ = “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject.

Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “$\times$” char
Turing Machine Example

\( M_1 \) accepts inputs in language \( B = \{ w\#w \mid w \in \{0,1\}^* \} \)

\( M_1 = \) “On input string \( w \):

1. Zig-zag across the tape to corresponding positions on either side of the \# symbol to check whether these positions contain the same symbol. If they do not, or if no \# is found, reject.

Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “\( \times \)” char

“Cross off” = write “\( \times \)” char
$M_1$ accepts inputs in language $B = \{w\#w | w \in \{0,1\}^*\}$

$M_1$ = “On input string $w$:
1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject.

Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “$\times$” char
Turing Machine Example

\( M_1 \) accepts inputs in language \( B = \{ w\#w \mid w \in \{0,1\}^* \} \)

\( M_1 = \) “On input string \( w \):

1. Zig-zag across the tape to corresponding positions on either side of the \# symbol to check whether these positions contain the same symbol. If they do not, or if no \# is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “x” char

“zag” to start
Turing Machine Example

$M_1$ accepts inputs in language $B = \{ w\#w \mid w \in \{0,1\}^* \}$

$M_1 =$ “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject.

Cross off symbols as they are checked to keep track of which symbols correspond.

“Cross off” = write “$\times$” char

Continue crossing off
Turing Machine Example

\( M_1 \) accepts inputs in language \( B = \{ w \# w | w \in \{0,1\}^* \} \)

\( M_1 = \) “On input string \( w \):

1. Zig-zag across the tape to corresponding positions on either side of the \( \# \) symbol to check whether these positions contain the same symbol. If they do not, or if no \( \# \) is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

2. When all symbols to the left of the \( \# \) have been crossed off, check for any remaining symbols to the right of the \( \# \). If any symbols remain, reject; otherwise, accept.”
Turing Machine Example

$M_1$ accepts inputs in language $B = \{ w\#w \mid w \in \{0,1\}^* \}$

$M_1 =$ “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are checked to keep track of which symbols correspond.

2. When all symbols to the left of the # have been crossed off, check for any remaining symbols to the right of the #. If any symbols remain, reject; otherwise, accept.”
Definition 3.3

A Turing machine is a 7-tuple, \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})\), where \(Q, \Sigma, \Gamma\) are all finite sets and

1. \(Q\) is the set of states,
2. \(\Sigma\) is the input alphabet not containing the blank symbol \(\square\),
3. \(\Gamma\) is the tape alphabet, where \(\square \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
4. \(\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}\) is the transition function,
5. \(q_0 \in Q\) is the start state,
6. \(q_{\text{accept}} \in Q\) is the accept state, and
7. \(q_{\text{reject}} \in Q\) is the reject state, where \(q_{\text{reject}} \neq q_{\text{accept}}\).
Formal Turing Machine Example

\[ B = \{w\#w | w \in \{0,1\}^*\} \]

- Move Right until #
- Accept if all crossed out
- Cross off 0 or 1
- "zag" Left to last x
- Move Right until #
- Read char (0 or 1), cross it off, move head R(ight)
Turing Machine: Informal Description

• $M_1$ accepts if input is in language $B = \{w#w \mid w \in \{0,1\}^*\}$

$M_1$ = “On input string $w$:

1. Zig-zag across the tape to corresponding positions on either side of the # symbol to check whether these positions contain the same symbol. If they do not, or if no # is found, reject. Cross off symbols as they are scanned to keep track of which symbols correspond.

2. When all symbols to the right of the # are crossed off, check for any remaining symbols to the left of the #. If any symbols remain, reject; otherwise, accept.”

We will (mostly) stick to informal descriptions of Turing machines, like this one.
TM Informal Description: Caveats

• TM informal descriptions are not a “do whatever” card
  • They must be sufficiently precise to communicate the formal tuple

• Input must be a string, written with chars from finite alphabet

• An informal “step” represents sequence of formal transitions
  • I.e., some finite number of transitions
  • It cannot run forever
  • E.g., can’t say “try all numbers” as a “step”
Non-halting Turing Machines (TMs)

• A DFA, NFA, or PDA always halts
  • Because the (finite) input is always read exactly once

• But a Turing Machine can run forever
  • E.g., the head can move back and forth in a loop

• Thus, there are two classes of Turing Machines:
  • A recognizer is a Turing Machine that may run forever
  • A decider is a Turing Machine that always halts.

**Definition 3.5**
Call a language *Turing-recognizable* if some Turing machine recognizes it.

**Definition 3.6**
Call a language *Turing-decidable* or simply *decidable* if some Turing machine decides it.
Formal Definition of an “Algorithm”

• An algorithm is equivalent to a Turing-decidable Language
Check-in Quiz 3/10

On Gradescope