The Post Correspondence Problem

Seo Yeon (Gloria) Chun and Alicia Li

June 1, 2021

Contents

1 Introduction

First introduced by Emil Post in 1946, the Post Correspondence Problem (*PCP*) is a well-known example of an undecidable decision problem. The goal of the problem is to determine whether or not a given set of dominos

$$\left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\}$$

has a *match*, which means that dominos from the set can be arranged in a way (with repetition) such that the top and the bottom strings are identical. In this paper, we show that there is no algorithm that decides whether a given instance of *PCP* has a

match, following the approach from [1]. To prove this result, we first introduce a method of computation: the Turing machine.

A decade before Post introduced *PCP*, mathematician Alan Turing defined a computational model called the Turing machine. In Section 4, we examine how Turing machines simulate algorithms.

To simplify the task of proving that *PCP* is undecidable, we construct a *reduction* from the undecidable problem A_{TM} (Section 5), which considers whether or not a certain Turing machine *accepts* a given input string.

After proving the undecidability of *PCP* (Section 7), we also apply this result to other problems such as determining whether or not a *context-free language* (Section 8) is *ambiguous* and if two context-free languages contain an overlapping string (Section 9).

2 Languages

In this section, we give formal definitions of a language and its properties.

Definition 2.1. First of all, an *alphabet* is any nonempty finite set and its members are called *symbols*. A finite sequence of symbols is called a *string*.

Typically, we use the variables Σ and Γ to denote alphabets.

Definition 2.2. A *language* is a set of strings. If A is a set of strings that some machine M accepts, we say that A is the language of machine M and express it as $L(M) = A$. We can also say that M *accepts* or *recognizes* A.

For a language A, we also define the *star* $*$ operation as

$$A^* = \{x_1 x_2 \cdots x_k \mid k \geq 0 \text{ and each } x_i \in A\}.$$

3 *PCP*

In this section, we introduce the *Post Correspondence Problem* (*PCP*). The problem can be easily defined as a puzzle:

We are given a collection of dominos, each of which contains a string on each side (top and bottom), as shown below:

$$\left\{\left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right]\right\}.$$

Our task is to determine whether a match exists.

Definition 3.1. A *match* is a sequence of dominos $i_1, i_2, \ldots, i_\ell$ such that the string generated by reading off all the symbols on the top side is identical to the string of symbols on the bottom:

$$t_{i_1} t_{i_2} \cdots t_{i_l} = b_{i_1} b_{i_2} \cdots b_{i_\ell}.$$

Definition 3.2. We define *PCP* to be the language of collections of dominos with such a match:

$$PCP = \{\langle P\rangle \mid \text{the set } P \text{ of dominos has a match}\}.$$

Here, we use the notation $\langle\cdot\rangle$ to denote a string encoding of the input.

Example 3.3. For the puzzle $\left\{\left[\frac{b}{ca}\right], \left[\frac{a}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{abc}{c}\right]\right\}$, the following is a match:

$$\left[\frac{a}{ab}\right], \left[\frac{b}{ca}\right], \left[\frac{ca}{a}\right], \left[\frac{a}{ab}\right], \left[\frac{abc}{c}\right].$$

Both the top and bottom strings read "$abcaaabc$," so the above sequence of dominos is a match.

For some combination of dominos, however, finding a match may not be possible.

Example 3.4. The collection $\left\{\left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right], \left[\frac{abc}{ab}\right], \left[\frac{ca}{c}\right], \left[\frac{acc}{ba}\right]\right\}$ cannot contain a match since every top string is longer than its corresponding bottom string.

Now, it would be convenient to be able to find a method to determine which collections are in *PCP* and which ones aren't. However, *PCP* is *undecidable*, meaning that there is no programmable algorithm that can determine if a match exists. To prove this, we must first explain our method of computation: Turing machines.

4 Turing Machines

In this section, we define Turing machines, the model of computation that we use for the rest of the paper. A Turing machine is a machine that recognizes a language. It has a finite set of states, transitions, and an infinite tape (unlimited memory).

More precisely, we will define a Turing machine as the following tuple:

Definition 4.1. A *Turing machine* is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where

1. Q is the finite set of *states*;
2. Σ is the *input alphabet* not containing the *blank symbol* $\sqcup$;
3. Γ is the *tape alphabet*, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$;
4. $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ 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* and $q_{\text{reject}} \neq q_{\text{accept}}$.

The Turing machine uses a tape head to navigate right or left and to read or edit the elements on the tape. Initially, the tape contains only the input, but as the Turing machine computes, the head may edit the elements on the tape.

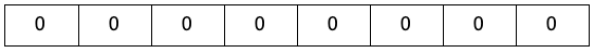
For instance, when the machine is in state q and the tape head is located over the symbol a, if $\delta(q, a) = (r, b, X)$, then the machine replaces a with b, enters state r, and moves in direction X (either left L or right R) along the tape.

Definition 4.2. If the Turing machine reaches either the accept or reject state, computation is completed, and we say that the Turing machine *halts* on this input. If computation is never completed, however, we say that the Turing machine *loops*.

Definition 4.3. A language is *recognizable* if there exists a Turing machine which will only halt and accept for the strings in its language and either reject or not halt at all for strings not in the language.

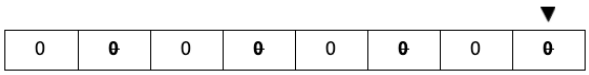
Definition 4.4. A language is *decidable* if there is a Turing machine which halts on every input string. These types of Turing machines are also called *deciders*.

Here, we give an example of a Turing machine and its state diagram that recognizes the language $B = \{0^{2^n} | n \in \mathbb{Z}_{\geq 0}\}$. The language B would accept inputs such as the following:



Example 4.5. The Decider M for the above Turing machine would work as follows. When given the input above, for instance, it would

1. Move left to right across the tape, crossing out every second 0.



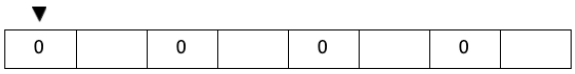
2. Accept the tape if it held a single 0 in step 1.

   Initially, the machine would skip this step since it held or crossed out four zeros, not one.

3. Reject if the tape held more than one 0 in step 1 and the number of 0s was odd.

   Initially, the machine would skip this step as well since it held or crossed out four zeros, which is an even number.

4. Return the tape head all the way to the tape's left end.



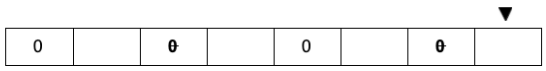
5. Return back to step 1, then repeat until the machine halts.

   In this case, the machine would come to a stop when it crosses out a single 0 and moves on to step 2, where it would accept the input string.

Below are the next two repetitions of the four steps above.

The second repetition:

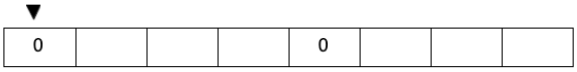
2.1 Move left to right across the tape crossing out every second 0.



2.2 Skip. The machine holds two 0s.

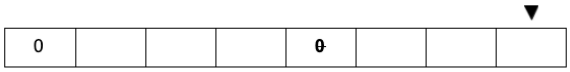
2.3 Skip. The machine holds two 0s.

2.4 Return the tape head to the tape's left end.



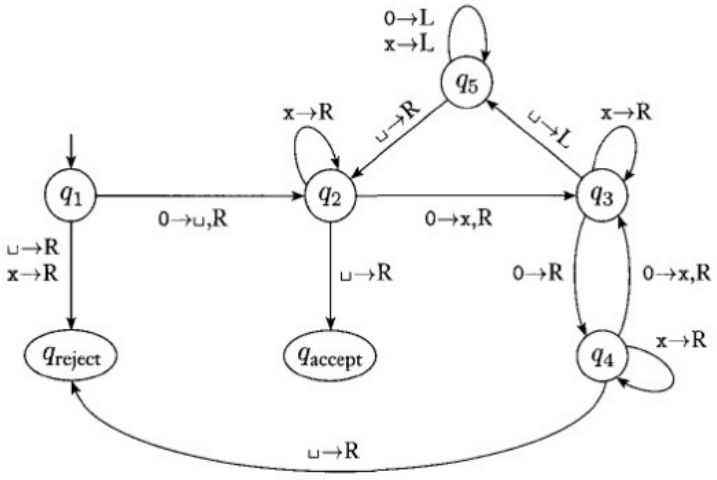
The third (and last) repetition:

3.1 Move left to right across the tape, crossing out every second 0.



3.2 Accept; machine halts.

Below is a state diagram that also represents the process above:



State diagram for the language $B = \{0^{2^n} | n \in \mathbb{Z}_{\geq 0}\}$ [p. 6 of [2]]

In the state diagram, the arrow pointing to state q_1 indicates that it is the start state, and the arrows connecting state to state give the transition function. The label $a \to b, X$ of a transition arrow indicates that the tape head reads the symbol a and replaces it by b. The tape head then moves in direction X, either left (L) or right (R). Some labels only say $a \to X$, meaning that the tape head does not edit the symbol before moving to its new location.

Definition 4.6. A *configuration* of a Turing machine is a setting of three elements: the current state, the current tape contents, and the current head location. Changes occur in these elements during Turing machine's computation process.

We represent a configuration in the order uqv where uv is the current tape contents, q is the current state, and the first symbol of v is the current location of the tape head.

Definition 4.7. We define the *computation history* of a Turing machine on a given input to be the sequence of such configurations.

Example 4.8. For instance, the computation history for the Turing Machine that accepts $B = \{0^{2^n} | n \in \mathbb{Z}_{\geq 0}\}$ (Example 4.5) when given the input string of 00 is

$$q_1 00,\ \sqcup q_2 0,\ \sqcup x q_3 \sqcup,\ \sqcup q_5 x \sqcup,\ q_5 \sqcup x \sqcup,\ \sqcup q_2 x \sqcup,\ \sqcup x q_2 \sqcup,\ \sqcup x \sqcup q_{\text{accept}}.$$

We are also able to use the brief descriptions of algorithms in lieu of formal Turing machine definitions by the Church-Turing Thesis.

Church-Turing Thesis. The *Church-Turing Thesis* states that the intuitive notion of algorithms (Church's definition of algorithms) is equivalent to Turing machine algorithms (Turing's definition of algorithms).

5 Proving A_{TM} is Undecidable

In this section, we will define the language A_{TM}, give its Turing machine, and prove that it is not decidable.

Definition 5.1. We define the language A_{TM} to be

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts string } w\}.$$

Theorem 5.2. *The language A_{TM} is recognizable.*

Proof. We construct a Turing machine K that recognizes A_{TM}: on input $\langle M, w\rangle$,

1. K simulates M on input w.
2. If M accepts, K accepts. □

Theorem 5.3. *The language A_{TM} is undecidable.*

Proof. Assume, for the sake of contradiction, that A_{TM} is decidable. Then by the definition of decidable languages (Definition 4.4), there must exist some decider H with $L(H) = A_{TM}$, i.e., for an input string $\langle M, w\rangle$,

1. H accepts if M accepts w, and
2. H rejects if M rejects w.

Now, consider a new Turing machine D, which takes Turing machines $\langle M \rangle$ as inputs. The machine D also simulates the decider H on input $\langle M, \langle M \rangle \rangle$: in particular, the machine D

1. accepts when H rejects $\langle M, \langle M \rangle \rangle$, and
2. rejects when H accepts $\langle M, \langle M \rangle \rangle$.

In other words, D rejects when H accepts and D accepts when H rejects.

Consider a case when D takes in itself, $\langle D \rangle$, as an input. Then by construction, D will

1. reject $\langle D \rangle$ when H accepts $\langle D, \langle D \rangle \rangle$, and
2. accept $\langle D \rangle$ when H rejects $\langle D, \langle D \rangle \rangle$.

However, because H accepts $\langle D, \langle D \rangle \rangle$ if and only if D accepts $\langle D \rangle$, we see that there exists a contradiction either way:

1. D rejects $\langle D \rangle$ if and only if D accepts $\langle D \rangle$, or
2. D accepts $\langle D \rangle$ if and only if D rejects $\langle D \rangle$.

Therefore, such a decider H cannot exist, so A_{TM} is undecidable. □

6 Mapping Reducibility

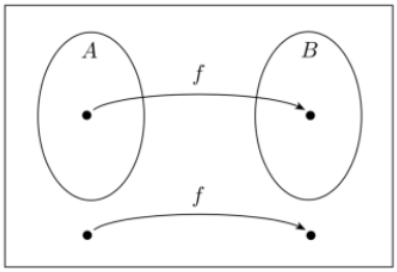
In this section, we introduce the concept of mapping reducibility, a method to prove that problems are computationally undecidable.

Definition 6.1. A function $f : \Sigma^* \to \Sigma^*$ is *computable* if there exists a Turing machine that halts with $f(w)$ on its tape for every input $w \in \Sigma^*$.

Definition 6.2. A language A is *mapping reducible* to a language B if there exists a computable function $f : \Sigma_A^* \to \Sigma_B^*$ such that for every $w \in \Sigma_A^*$, $w \in A$ if and only if $f(w) \in B$.

This is written as $A \leq_m B$, and the function f is called the *reduction* from A to B.

Below is an illustration of mapping reducibility.



Mapping reducibility illustration, [p. 235 of [1]].

In computability theory, reducibility helps us classify problems by their decidabilities. When proving that a certain language is undecidable, for instance, we can simply reduce the language from another language that is already known or proven to be undecidable. Below is a theorem that captures this idea.

Theorem 6.3. *Let A and B be languages. If $A \leq_m B$ and B is decidable, then A is decidable.*

Proof. Let M be the decider for B and f be the reduction from A to B. We construct a decider N for the language A, which works as follows: on any input w, it

1. Computes $f(w)$, and
2. Runs the decider M on input $f(w)$ and yields the same output as that of M.

Since $A \leq_m B$ and M is a decider, the machine M accepts $f(w)$ if $w \in A$ and rejects otherwise, which means that the machine N is a decider as well and executes as expected. □

Corollary 6.4. *If $A \leq_m B$ and A is undecidable, then B is undecidable.*

7 *PCP* is Undecidable

Our strategy for the proof is to reduce A_{TM} to another language called Modified PCP (*MPCP*) and then reduce *MPCP* to *PCP*. Since we already know that A_{TM} is undecidable, it would follow that *MPCP*, and thus *PCP*, are both undecidable as well (Corollary 6.4). Now all we need to do is construct computable functions from A_{TM} to *MPCP* and *MPCP* to *PCP* to prove their reducibilities.

First, we define *Modified PCP* (*MPCP*):

Definition 7.1. We define *MPCP* to be the language

$$MPCP = \left\{ \langle P' \rangle \mid P' \text{ is a collection of dominos with a match starting with the first domino } \left[\frac{t_1}{b_1}\right] \right\}.$$

Lemma 7.2. *We have $A_{TM} \leq_m MPCP$.*

Proof. Given a Turing machine M and an input string $w = w_1 w_2 \ldots w_n$, we construct an instance P' of *MPCP* such that M accepts w if and only if $P' \in MPCP$. The reduction essentially simulates an accepting computation through an *MPCP* match.

Step 1: First, we insert the following domino into P' as the first in the match ($\left[\frac{t_1}{b_1}\right]$):

$$\left[\frac{\#}{\#q_0 w_1 w_2 \ldots w_n}\right],$$

where $q_0 w_1 w_2 \ldots w_n$ is the first configuration in the computation history of M on input w.

So far, we only have a # on the top string of the match. In steps 2,3, and 4 we will add more dominos to simulate parts of the computation: transitions to the right, transitions to the left, and symbols that stay the same, respectively.

Step 2: Here, we deal with tape head movements to the right.

For every $a, b \in \Gamma$ and $q, r \in Q$ ($q \neq q_{\text{reject}}$), we insert the domino

$$\left[\frac{qa}{br}\right]$$

into P' if $\delta(q, 0) = (r, b, R)$.

Step 3: Here, we deal with tape head movements to the left.

For every $a, b, c \in \Gamma$ and $q, r \in Q$ ($q \neq q_{\text{reject}}$), we insert the domino

$$\left[\frac{cqa}{rcb}\right]$$

into P' if $\delta(q, a) = (r, b, L)$.

Step 4: Here, we add the tape alphabet.

For every $a \in \Gamma$, insert $\left[\frac{a}{a}\right]$ into P'.

These symbol dominos allow us to construct the part of the configuration not included in the transition function.

Step 5: Now we insert $\left[\frac{\#}{\#}\right]$ and $\left[\frac{\#}{\sqcup\#}\right]$ into P' to separate configurations.

The second domino with the #s allows us to simulate the infinite number of $\sqcup$ symbols that are not considered in the method of representing configurations. Note that if there is an empty string ϵ, we replace it with a $\sqcup$.

Step 6: For every $a \in \Gamma$, insert $\left[\frac{aq_{\text{accept}}}{q_{\text{accept}}}\right]$ and $\left[\frac{q_{\text{accept}}a}{q_{\text{accept}}}\right]$ into P'.

This accounts for the steps after the Turing Machine has halted, when it "eats," or cancels out, the remaining symbols on the tape until there is nothing left.

Step 7: The final step is to add the the domino

$$\left[\frac{q_{\text{accept}}\#\#}{\#}\right]$$

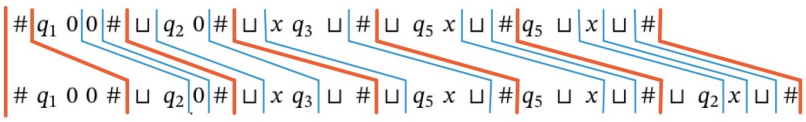
in order to even out the top and bottom sides and finish constructing P'. Note that this will also complete the match.

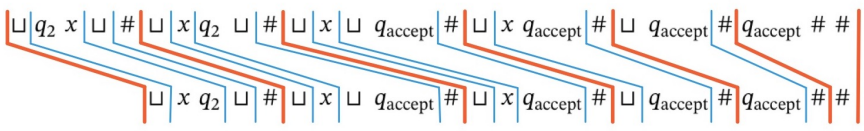
We now verify that this is a valid reduction. If M accepts w, we have an accepting computation history. Using this computation history, we can construct an $MPCP$ match for P', where the top and bottom strings will start with the configuration history separated by #s and end by "eating" the symbols remaining on the tape. Our match starts with the first domino. To choose subsequent dominos, we slice each configuration into substrings, inserting transition dominos for the sections of the tape that change in successive configurations and $\left[\frac{a}{a}\right]$ dominos for symbols that stay the same in successive configurations. After reaching an accepting configuration, we continue to slice the configuration into substrings, this time adding "eating" dominos to delete symbols next to the head and $\left[\frac{a}{a}\right]$ dominos for the rest of the symbols. We repeat the deletion process until only q_{accept} remains and finish the match by adding the last domino (from Step 7).

Conversely, if there is a $MPCP$ match for P', then we show that A_{TM} accepts $\langle M, w\rangle$. We design our match to start with the Step 1 domino, and it must end with the Step 7 domino, as this is the only way for the #s to align. Since we start with the Step 1 domino, which contains the first configuration in the bottom string, the top string of the match must start with the first configuration as well. This part of the top string is comprised of the same dominos as the next part of the bottom string, and the match links the unchanging part of each configuration to the next by using symbol dominos, while linking altering components with transition dominos or "eating" dominos. Hence, each configuration must either (i) be the result of applying the transition function to the previous configuration, or (ii) be the result of "eating" a symbol next to q_{accept} in the previous configuration. However, since "eating" dominos cannot be used until an accept state is reached, they do not affect computation. Therefore, our match must yield an accepting configuration. □

The following is an example of the reduction, giving the $MPCP$ match corresponding to the accepting computation history in Example 4.8.

Example 7.3. We use our Turing machine for the language $B = \{0^{2^n} \mid n \in \mathbb{Z}_{\geq 0}\}$ on the input 00.





The first domino in the match is from Step 1; the second is a transition from Step 2; the next domino is a symbol domino from Step 4; then we have a configuration separator from Step 5. The match simulates computation since each configuration goes to the next through the domino. Note the configuration separator with $\sqcup$ in the transition from the second to third configurations. We end with several "eater" dominos from Step 6, and ultimately, the domino from Step 7.

Now we must reduce *MPCP* to *PCP* to complete the proof. *MPCP* is a necessary intermediary step because if we did not require for the match to start with the specified domino $\left[\frac{t_1}{t_2}\right]$, the tape alphabet domino $\left[\frac{a}{a}\right]$ could also be a match, which would not simulate an accepting computation history.

Lemma 7.4. *We have $MPCP \leq_m PCP$.*

Proof. We can force the match to begin with the specified domino $\left[\frac{t_1}{b_1}\right]$ through the following construction:

We first define the $\star$ operation as follows:

$$\begin{aligned} \star u &= * \, u_1 * u_2 * u_3 \cdots * u_n, \\ u\star &= u_1 * u_2 * u_3 \cdots * u_n *, \\ \star u\star &= * \, u_1 * u_2 * u_3 \cdots * u_n * . \end{aligned}$$

Given an instance

$$P' = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \left[\frac{t_3}{b_3}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$$

of *MPCP*, we can construct our instance of *PCP* to be

$$P = \left\{ \left[\frac{\star t_1}{\star b_1 \star}\right], \left[\frac{\star t_2}{b_2 \star}\right], \left[\frac{\star t_3}{b_3 \star}\right], \ldots, \left[\frac{\star t_k}{b_k \star}\right], \left[\frac{* \, \diamondsuit}{\diamondsuit}\right] \right\},$$

where $\left[\frac{*\diamondsuit}{\diamondsuit}\right]$ accounts for the extra $*$ needed to complete the match.

If P' has a match $i_1 = 1, i_2, \ldots, i_\ell$, then the match in the above construction P is $i_1, i_2, \ldots, i_\ell, k+1$, where the $(k+1)$th domino is $\left[\frac{*\diamondsuit}{\diamondsuit}\right]$.

Conversely, if P has a match $j_1, j_2, \ldots, j_m$, it must start with $j_1 = 1$ because $\left[\frac{\star t_1}{\star b_1 \star}\right]$ is the only one where the bottom string starts with $*$. Moreover, the last domino must be $\left[\frac{*\diamondsuit}{\diamondsuit}\right]$ because an extra $*$ is needed at the end of the top string. Therefore, $j_1, \ldots, j_{m-1}$ is a match for P' starting with the specified first domino. □

Theorem 7.5. *The language PCP is undecidable.*

Proof. By Lemmas 7.2 and 7.4, we have reductions from A_{TM} to *MPCP* and *MPCP* to *PCP*. Since A_{TM} is undecidable by 5.3, Corollary 6.4 shows that *PCP* is undecidable as well. □

Next, we will reduce *PCP* to other languages in order to show that they are undecidable as well. To define these languages, we will introduce context-free languages and grammars.

8 Context-Free Languages

In this section, we describe context-free grammars and context-free languages. Each *context-free grammar* has a collection of *substitution rules* in the format $A \to \alpha$ in which the symbol A on the left is a *variable*, which we represent by capital letters, and the string α on the right is comprised of variables as well as other symbols called *terminals*, which we represent by lowercase letters.

Definition 8.1. A *context-free grammar* is a tuple (V, Σ, R, S) where

1. V is the finite set of *variables*, which we typically represent using capital letters;
2. Σ is the finite set of *terminals* (separate from V), which we typically represent by lowercase letters;
3. R is the finite set of *substitution rules* of the form $A \to \alpha$, where $A \in V$ and α is a string of variables and terminals; and
4. $S \in V$ is the start variable.

To generate a string through a grammar, we go through the following steps:

1. Write the *start variable*, which we typically write as the left symbol of the top-most rule.
2. Replace a variable with the right hand side of a corresponding rule.
3. Repeat the second step until there are no more variables.

Example 8.2. Consider the following context-free grammar:

$$
\begin{aligned}
\text{[SENTENCE]} &\to \text{[SUBJECT][PREDICATE]} \\
\text{[SUBJECT]} &\to \text{BOB} \mid \text{GRU} \\
\text{[PREDICATE]} &\to \text{[VERB]} \mid \text{[SPEECH-PHRASE]} \\
\text{[VERB]} &\to \text{VISITED} \mid \text{FLEW} \mid \text{[VERB][ADVERB]} \\
\text{[SPEECH-PHRASE]} &\to \text{[SPEECH-VERB][SENTENCE]} \\
&\quad \mid \text{[SPEECH-VERB][SENTENCE][ADVERB]} \\
\text{[SPEECH-VERB]} &\to \text{SAID} \mid \text{SHOUTED} \\
\text{[ADVERB]} &\to \text{YESTERDAY} \mid \text{QUICKLY} \mid \text{FREQUENTLY}
\end{aligned}
$$

We denote separate substitution possibilities that come from the same variable with bars (|).

From this grammar, we can form the *ambiguous* sentence "Bob said Gru visited yesterday." There are two ways to interpret this:

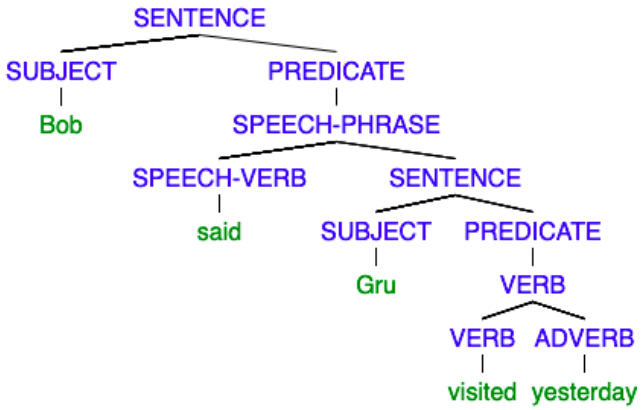
1. Gru visited yesterday, said Bob.

2. Bob said yesterday that Gru visited.

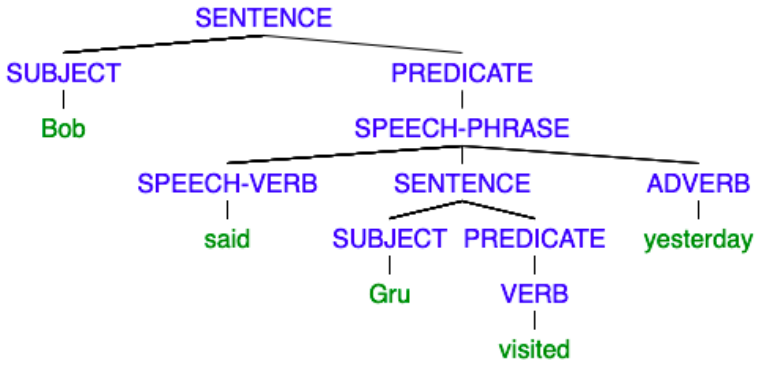
Hence, depending on whether the word "yesterday" is interpreted as an adverb modifying "visited" or "said," two different messages can be delivered to the readers.

We can also represent this sequence of steps with a diagram called a *parse tree*. Below are the two parse trees that represent these possibilities:

1. Parse Tree 1: "Gru visited yesterday, said Bob."



2. Parse Tree 2: "Bob said yesterday that Gru visited."



Definition 8.3. A context-free grammar that can generate the same string in more than one way is said to be *ambiguous*.

Going back to Example 8.2, the sentence "Bob said Gru visited yesterday" would have an ambiguous context-free grammar since both trees generate the same sentence, but in different ways.

Definition 8.4. A *context-free language* is the set of strings generated by a context-free grammar.

9 $AMBIG_{CFG}$ and $OVERLAP_{CFG}$ are Undecidable

The language $AMBIG_{CFG} = \{\langle C \rangle : C \text{ is an ambiguous CFG}\}$ is undecidable. To show this, we create a reduction from PCP to $AMBIG_{CFG}$ by letting the top string of the match be one of the derivations and the bottom string be the second derivation of the ambiguous string.

Theorem 9.1. *The language $AMBIG_{CFG}$ is undecidable.*

Proof. We show that $PCP \leq_m AMBIG_{CFG}$. The reduction is the computable function $f : \Sigma^*_{PCP} \to \Sigma^*_{AMBIG_{CFG}}$ that sends the instance

$$P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \dots, \left[\frac{t_k}{b_k}\right] \right\}$$

of PCP to the context-free grammar C with the following set of rules:

$$\begin{aligned} S &\to T \mid B \\ T &\to t_1 T a_1 \mid \cdots \mid t_k T a_k \mid t_1 a_1 \mid \cdots \mid t_k a_k \\ B &\to b_1 B a_1 \mid \cdots \mid b_k B a_k \mid b_1 a_1 \mid \cdots \mid b_k a_k. \end{aligned}$$

If P has a match $i_1, i_2, \dots, i_k$, then C is ambiguous because there are two possible derivations for the string $t_{i_1} t_{i_2} \cdots t_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_1} = b_{i_1} b_{i_2} \cdots b_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_1}$:

1. By following the rule $S \to T$ and substituting T for $t_{i_n} T a_{i_n}$ in the $(n+1)$th substitution, we can construct the first derivation $t_{i_1} t_{i_2} \cdots t_{i_{k-1}} t_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$.
2. Similarly, by following the rule $S \to B$, we can construct the second derivation $b_{i_1} b_{i_2} \cdots b_{i_{k-1}} b_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$.

Conversely, suppose that C is ambiguous. Given a word and the first substitution $S \to T$ or $S \to B$ in its derivation, we can determine the rest of its derivation by examining the substring $a_{i_1} a_{i_2} \cdots a_{i_k}$ at the end. If we have an ambiguous word, the two derivations would look like

$$t_{i_1} t_{i_2} \cdots t_{i_{k-1}} t_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1} = b_{i_1} b_{i_2} \cdots b_{i_{k-1}} b_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}.$$

Since $a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1} = a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$, then $t_{i_1} t_{i_2} \cdots t_{i_{k-1}} t_{i_k} = b_{i_1} b_{i_2} \cdots b_{i_{k-1}} b_{i_k}$, and this string is our match. □

Similarly, we can prove that the context-free language

$$OVERLAP_{CFG} = \{\langle \mathcal{G}, \mathcal{H} \rangle \mid \mathcal{G} \text{ and } \mathcal{H} \text{ are CFGs where } L(\mathcal{G}) \cup L(\mathcal{H}) \neq \emptyset\}$$

is undecidable.

Theorem 9.2. *The language $OVERLAP_{CFG}$ is undecidable.*

Proof. We show that $PCP \leq_m OVERLAP_{CFG}$. The reduction is the computable function $f : \Sigma^*_{PCP} \to \Sigma^*_{OVERLAP_{CFG}}$ where given the instance $P = \left\{ \left[\frac{t_1}{b_1}\right], \left[\frac{t_2}{b_2}\right], \ldots, \left[\frac{t_k}{b_k}\right] \right\}$ of PCP, we construct the following CFGs $\mathcal{G}$ and $\mathcal{H}$:

$$\mathcal{G} : G \to t_1 G a_1 \mid \cdots \mid t_k G a_k \mid t_1 a_1 \mid \cdots \mid t_k a_k$$
$$\mathcal{H} : H \to b_1 H a_1 \mid \cdots \mid b_k H a_k \mid b_1 a_1 \mid \cdots \mid b_k a_k.$$

If P has a match $i_1, i_2, \ldots, i_k$, we can construct a common string in both the languages of $\mathcal{G}$ and $\mathcal{H}$ by substituting G for $t_{i_n} G a_{i_n}$, and H for $b_{i_n} H a_{i_n}$ in the nth substitution. The string we obtain is

$$t_{i_1} t_{i_2} \cdots t_{i_{k-1}} t_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1} = b_{i_1} b_{i_2} \cdots b_{i_{k-1}} b_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$$

which satisfies the requirements of $OVERLAP_{CFG}$.

Conversely, if $\mathcal{G}$ and $\mathcal{H}$ are overlapping languages, we can then create a match since the overlapping word can be expressed as both $t_{i_1} t_{i_2} \cdots t_{i_{k-1}} t_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$ and $b_{i_1} b_{i_2} \cdots b_{i_{k-1}} b_{i_k} a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$ for some indices $i_1, \ldots, i_k$ because the substring $a_{i_k} a_{i_{k-1}} \cdots a_{i_2} a_{i_1}$ is the same in both derivations. Then $i_1, \ldots, i_k$ is our match. □

10 Acknowledgements

We would like to thank MIT PRIMES Circle and its sponsors for providing us with this amazing opportunity and Alicia's younger brother, Aiden, for his moral support and for gracing us with occasional brain breaks.

Above all, a huge thank you to our mentor, Alexandra Hoey, for the weekly meetings, presentation tips, and patient explanations of the numerous computational terms, theorems, proofs, and more.

References

[1] Sipser, M. (2013). *Introduction to the theory of computation* (3rd ed.). Cengage Learning.

[2] Carnegie Mellon University. (n.d.). *State diagram* [Illustration]. https://www.cs.cmu.edu/~arielpro/15251/Lectures/lecture22.pdf

[3] Shang, M. (2011). *Syntax tree generator* [Computer software]. http://mshang.ca/syntree