Recall that a decision problem is a parameterized problem with a yes/no answer for each instance.

We say a decision problem is *solvable* if there is an algorithm (that is, a Turing machine) that unerringly solves it.

Some decision problems about CFL’s are solvable.

For example, “Given a CFG $G$, is $L(G) = \emptyset$?”

To solve it, apply the algorithm for removing all useless symbols from $G$. If any symbols remain, then $L(G) \neq \emptyset$. 
Unfortunately, many of the problems that we really want to solve about CFL’s are unsolvable. These include

- Given CFG $G = (V, \Sigma, P, S)$, is $L(G) = \Sigma^*$?
- Given CFG’s $G_1$ and $G_2$, is $L(G_1) = L(G_2)$?
- Given CFG’s $G_1$ and $G_2$, is $L(G_1) \cap L(G_2) = \emptyset$?
- Given CFG $G$, is $L(G)$ regular?
- Given CFG $G$, is $G$ ambiguous?
A configuration summarizes all the information needed to continue the execution of a Turing machine. It includes the current state, the current contents of the tape, and an indication of which cell is currently being scanned. Formally, a configuration is of the form $xqy$, where

- $xy$ is the current contents of the tape up to the last unreached cell;
- $q$ is the current state; and
- the TM is currently scanning the first symbol of $y$.

Note: there are other ways to specify a configuration. What is important is that one can computably extract the three pieces of information from the specification.
A valid computation of a TM is a list of configurations, corresponding to consecutive moves of the TM, starting with the initial configuration, and ending in the halting state.

The obvious way to encode a valid computation is

\[ C_0 \# C_1 \# C_2 \# \cdots \# C_n, \]

where each \( C_i \) is a configuration and \( \# \) is a new delimiter symbol.

However, we are free to encode it anyway we like, provided we can computably recover all the information.
Comparing consecutive configurations

Notice that $C_{i+1}$ differs from $C_i$ in at most three positions next to each other: the symbol that is changed, the state, and the movement of the state:

\[
\ldots qA \ldots \text{ becomes } \ldots Br \ldots \quad \text{(right move)}
\]
\[
\ldots AqB \ldots \text{ becomes } \ldots rAC \ldots \quad \text{(left move)}.
\]

Our goal is to permit a PDA to check whether a computation is valid or invalid.

But a PDA can’t compare two consecutive configurations because

\[
\{xx : x \in \{0, 1\}^*\}
\]

is not a CFL.

If we change the definition of computation, however, then a PDA can check it.
Redefining a computation

Here’s how: we redefine a computation to be

\[ C_0 \# C_1^R \# C_2 \# C_3^R \cdots \# C_{n-1} \# C_n, \]

if \( n \) is even and

\[ C_0 \# C_1^R \# C_2 \# C_3^R \cdots \# C_{n-1} \# C_n^R, \]

if \( n \) is odd.

Now a PDA can check if \( C_i \# C_{i+1}^R \) corresponds to two consecutive moves of a TM.

For example, if a “right” move is made by the TM, we have to check that \( C_i = xqAy \) and \( C_{i+1}^R = y^R rBx^R \) and \( \delta(q, A) = (r, B, R) \) is a move of the TM.
Checking a computation

To do so we push symbols on the stack until we see a state $q$ in the input. Then we store $q$ and the symbol $A$ that follows, and “look up” $\delta(q, A) = (r, B, R)$. We then push $rB$ onto the stack, and continue pushing symbols from the input onto the stack until $\#$ is seen. Then we pop the stack and compare it to the rest of the input.

Similarly for left and stationary moves.

Now we can prove:

**Theorem.** The following problem is unsolvable: given two PDA’s $M_1$ and $M_2$, does $L(M_1) \cap L(M_2) = \emptyset$?
The key idea is that *the set of valid computations of a TM can be written as the intersection of two CFL’s.*

**Theorem.** The following problem is unsolvable: given two PDA’s $M_1$ and $M_2$, does $L(M_1) \cap L(M_2) = \emptyset$?

**Proof.** We will reduce from the following decision problem: given a TM $M$, is $L(M) = \emptyset$? You should remember from CS 360/365 that this decision problem is unsolvable.

Given the TM $M$, we create two PDA’s as above. On input a computation, $M_1$ checks $C_{2i+1} \# C_{2i+2}$ is correct for all $i$; and $M_2$ checks that $C_{2i} \# C_{2i+1}^R$ is correct for all $i$. 
Furthermore, we arrange it so that $M_2$ also checks that the first configuration is an initial configuration of the form $q_0Bx$ for some $x$, and either $M_1$ or $M_2$ (depending on parity) checks that the final configuration is a halting configuration.

So $x \in L(M_1) \cap L(M_2)$ if and only if $x$ is a valid computation.

Thus $L(M_1) \cap L(M_2) = \emptyset$ iff $L(M) = \emptyset$.

This completes the reduction.

**Corollary.** The following decision problem is unsolvable: given CFG’s $G_1$ and $G_2$, is $L(G_1) \cap L(G_2) = \emptyset$?
Now we turn to a more interesting problem: the universality problem for PDA’s. This amounts to checking, for a PDA with input alphabet $\Delta$, whether $L(P) = \Delta^*$. 

We use the same idea with computations we used before. Now, though, instead of valid computations we focus on invalid computations: these are all strings that do not represent a valid computation.

The key insight is that the set of invalid computations of a TM is a CFL.
How can a string be an invalid computation?

▶ It could start wrong (something other than \( q_0 B x \# \)).
▶ It could end wrong (something other than \# y h z \)).
▶ It could have two consecutive \# symbols.
▶ It could be that if part of the input looks like \# y \# z \# then it is not the case that \( y = C_i, \ z = C_{i+1}^R \) (or \( y = C_i^R, \ z = C_{i+1} \), depending on parity)

We can make individual PDA’s for each of these. For the last one, we use two PDA’s, one that checks even \( i \) and one odd \( i \).

**Lemma.** The set of invalid computations of a TM is a CFL.
**Theorem.** The following problem is unsolvable: given a PDA $P$ with input alphabet $\Delta$, is $L(P) = \Delta^*$?

**Proof.** We reduce from the problem, given a TM $M$, is $L(M) = \emptyset$?

Given $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, h)$, we create a PDA $P$ as above, accepting the set of all invalid computations of $M$. This PDA $P$ has input alphabet $\Delta := \Sigma \cup \Gamma \cup \{#\}$.

Then every string in $\Delta^*$ is an invalid computation iff $L(M) = \emptyset$.

This completes the reduction.
Two corollaries

**Corollary.** The following problem is unsolvable: given a CFG $G = (V, \Delta, P, S)$, is $L(G) = \Delta^*$?

**Corollary.** The following problem is unsolvable: given two CFG's $G_1$ and $G_2$, is $L(G_1) = L(G_2)$?

*Proof.* Take $G_2$ to be the grammar that generates $\Delta^*$. 
**Theorem.** The following problem is unsolvable: given a CFG $G$, is $L(G)$ regular?

**Proof.** We reduce from the problem, given CFG $G$, is $L(G) = \Sigma^*$?

Given a CFG $G$, we create a CFG $G_2$ for the language

$$L_1 := L_0 \# \Sigma^* \cup \Sigma^* \# L(G),$$

where $L_0$ is any nonregular context-free language, such as

$$\{a^n b^n : n \geq 0\}.$$

We claim $L(G_2)$ is regular iff $L(G) = \Sigma^*$.

If $L(G) = \Sigma^*$, then $L_1 = \Sigma^* \# \Sigma^*$, which is regular.

If $L(G) \neq \Sigma^*$, then there must be a $w$ such that $w \notin L(G)$. Then $L_1/\#w = L_0$, which is not regular. So $L_1$ is not regular.
We have seen that the decision problem for CFG $G$,

Is $L(G)$ regular?

is not solvable.

But suppose we have a grammar $G$ and you know (somehow) that $L(G)$ is regular. Can you determine which regular language it is?

More formally, given $G$ such that $L(G)$ is regular, can one compute a DFA $A$ such that $L(G) = L(A)$?

This is an example of a “birdie” problem, because you imagine that you are given $G$ and a “little birdie” has told you that $L(G)$ is regular.
Theorem. The birdie problem: given $G$ and $M$, with $L(G)$ regular, compute DFA $A$ such that $L(G) = L(A)$, is unsolvable.

Proof. Let $M$ be any TM such that $L(M)$ is recursively enumerable but not recursive. (For example, we could take $M$ to recognize the language $\{\langle T \rangle : T$ is a TM such that $L(T) \neq \emptyset \}$.) Then there cannot be an algorithm that, for all $x$, decides whether $x \in L(M)$.

Assuming the birdie problem is solvable, we will produce such an algorithm. This will give us a contradiction.
We know there is a grammar $G' = (V, \Delta, P, S)$ such that $L(G')$ is the invalid computations of $M$, and we can compute this from $M$.

For each $x$, define $F_x = \{q_0By\#z : y \neq x \text{ and } z \in \Delta^*\}$.

Clearly each $F_x$ is regular, and we can easily compute a DFA $B_x$ recognizing $F_x$.

Given $B_x$ and $G'$, we can create a CFG $G_x$ such that $L(G_x) = \text{invalid}(M) \cup F_x$. 
We now claim $L(G_x)$ is regular:

- For if $x \in L(M)$, then $L(G_x) = \Delta^* - \{w\}$, where $w$ is the accepting computation of $M$ on $x$.
- If $x \not\in L(M)$, then $L(G_x) = \Delta^*$.

Now suppose there is a computable way to compute a DFA $A$ such that $L(A) = L(G_x)$. Given such a DFA, we can determine whether $L(A) = \Delta^*$ using standard techniques. Hence we can computably determine if $x \in L(M)$. 