5) Checking whether two given DFAs accept the same language
Given B_1, B_2 DFAs, test whether $L(B_1) = L(B_2)$.
We rewrite this problem as the language
$EQ_{DFA} = \{< B_1, B_2> \mid B_1 \text{ and } B_2 \text{ are DFAs and } L(B_1) = L(B_2)\}$.

Theorem: EQ_{DFA} is a Turing-decidable language.

Proof:
Given two sets Γ and Σ, $\Gamma \neq \Sigma$ if $\exists\ x \in M$ such that $x \notin \Sigma$ (i.e. $\Gamma \setminus \Sigma \neq \emptyset$) or $\exists\ x \in \Sigma$ such that $x \notin \Gamma$ (i.e. $\Sigma \setminus \Gamma \neq \emptyset$).
Recall from our unit on set theory that $\Gamma \setminus \Sigma = \Gamma \cap \sim\Sigma$, Γ intersect the complement of Σ.
Similarly, $\Sigma \setminus \Gamma = \Sigma \cap \sim\Gamma$.
Therefore, $\Gamma \neq \Sigma \Leftrightarrow (\Gamma \cap \sim\Sigma) \cup (\Sigma \cap \sim\Gamma) \neq \emptyset$. This expression is called the **symmetric difference** of sets Γ and Σ in set theory.

Now, returning to our problem, note that B_1 and B_2 are DFAs $\Rightarrow L(B_1)$ and $L(B_2)$ are regular languages. Furthermore, we showed that the set of regular languages is closed under union, intersection and the taking of complements $\Rightarrow (L(B_1) \cap \sim(L(B_2)) \cup (L(B_2) \cap \sim(L(B_1))$ is a regular language $\Rightarrow$ C a DFA that recognises the symmetric difference of $L(B_1)$ and $L(B_2)$ $(L(B_1) \cap \sim(L(B_2)) \cup (L(B_2) \cap \sim(L(B_1))$.
$L(B_1) = L(B_2)$ if this symmetric different is empty $\Rightarrow \forall <B_1, B_2> \in EQ_{DFA}\ \exists <C> \in E_{DFA}$, the language corresponding to the emptiness testing problem.
Since E_{DFA} is Turing-decidable, EQ_{DFA} is Turing-decidable. (*q.e.d*)

Next, we look at context-free grammars (CFGs) that we studied last term.

6) L_{CFG} = {<G, w> | G is a CFG and w is a string}
Theorem: L_{CFG} is a Turing-decidable language.

Sketch of Proof:
We could try to go through all possible applications of production rules allowable under G to see whether we can generate w, but infinitely many derivations may need to be tried. Therefore, if G does not generate w, our algorithm would not halt. We would thus have a Turing machine that is a recogniser but **not** a decider.
To get a decider we have to put G into a special form called a Chomsky normal form that takes 2n-1 steps to generate a string w of length n. We do not need to know what a Chomsky normal form is, just that one exists in order to write down our decider M.

M = on input <G, w>, where G is a context-free grammar and w is a string.
1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with 2n-1 steps, where n is the length of w if n > 0. If n = 0 list all derivations with one step.
3. If any of these derivations generate w then accept, otherwise reject.

7) Emptiness testing for context-free grammars

Given a context-free grammar G, figure out whether the language it generates L(G) is empty or not.

Rewrite as a language E_{CFG} = {<G> | G is a CFG and L(G) = Ø}

Theorem: E_{CFG} is a Turing-decidable language.

Proof:

We use a similar marking argument as we did to show E_{DFA} was Turing-decidable. We define the Turing machine as:

M = on input <G>, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked.
3. Mark any nonterminal <T> if G contains a production rule <T> ⇒ $u_1, \ldots, u_k$ has already been marked.
4. If the start symbol <S> is not marked then accept, otherwise reject.

As we can see from step 4, if <S> is marked then the context-free grammar will end up generating at least one string as all terminals have already been marked in step 1. Therefore, L(G) ≠ Ø and we reject G. (*q.e.d*)

8) Equivalence problem for context-free grammars

Given two context-free grammars G_1 and G_2, determine whether they generate the same language, i.e. $L(G_1) = L(G_2)$.

Rewrite this problem as a language:

EQ_{CFG} = {< G_1, G_2> | G_1 and G_2 are CFGs and $L(G_1) = L(G_2)$}'.

To solve the equivalence problem for DFAs, we used the symmetric difference and the fact that the emptiness problem for DFAs is Turing-decidable. In this case, the emptiness problem for CFGs is Turing-decidable as we just proved, but the symmetric difference argument does **not** work as the set of languages produced by context-free grammar is **not** closed under complements or intersection so the following result is true instead:

Proposition: EQ_{CFG} is **not** a Turing-decidable language.

This proposition is proven using a technique called reducibility. An even more general result is true, the equivalence problem for Turing machines is undecidable:

EQ_{TM} = {<M_1, M_2> | M_1 and M_2 are Turing machines and $L(M_1) = L(M_2)$}.

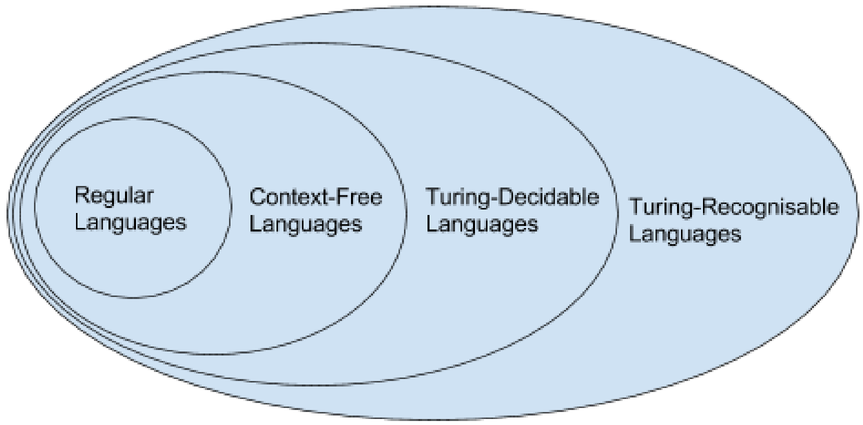
Proposition: E_{TM} is **not** a Turing-decidable language.

Returning to context-free grammars, we now know that L_{CFG} and E_{CFG} are Turing-decidable, but EQ_{CFG} is not.

Recall that a language is context-free if it can be generated by a context-free grammar.

Moral of the Story:
We know how the main types of languages relate to each other:
{Regular Languages} ⊂ {Context-Free Languages} ⊂ {Turing-Decidable Languages} ⊂ {Turing-Recognisable Languages}

Visually we represent the relationship with a Venn diagram:



So Turing machines provide a very powerful computational model. What is surprising is that once we have build a Turing machine to recognise a language, we do not know whether there is a simpler computational model such as a DFA that recognises the same language. Define:

Regular_{TM} = {<M> | M is a Turing machine and L(M) is a regular language}.

Theorem: Regular_{TM} is **not** a Turing-decidable language.
This theorem is proven using reducibility. In fact, even more is true:
Rice's Theorem: Any property of the languages recognised by Turing machines is not Turing-decidable.

Undecidability

Task: Understand why certain problems are algorithmically unsolvable.
Recall that a Turing machine is defined as a 7-tuple (S, A, Ã, t, i, S_{acc}, S_{rej}) where:

- S = Set of States
- A = Input Alphabet **not** containing the blank symbol _
- Ã = Tape Alphabet where _ ∈ Ã and A ⊆ Ã
- t = Transition Mapping t : S x Ã → S x Ã x {L, R}
- i = Initial State
- S_{acc} = Accept State
- S_{rej} = Reject State

Definition: An **encoding** <M> of a Turing machine M refers to the 7-tuple (S, A, Ã, t, i, S_{acc}, S_{rej}) that defines M and is therefore a finite string.
Recall that earlier in the module we proved the following results:

Theorem:
If A is a finite alphabet, then the set of all words over A ($A^* = A^0 \cup A^1 \cup \ldots \cup A^\infty$) is countably infinite.

Corollary 1:
If A is a finite alphabet, then the set of all languages over A is uncountably infinite.

Corollary 2:
The set of all programs in any programming language is countably infinite.
Recall that we proved *corollary 2* by realising that for any programming language, a program is a finite string over the finite alphabet of all allowable character in that programming language.

Corollary 3:
Given a finite alphabet A, the set of all Turing-recognisable languages over A is countably infinite.

Proof:
An encoding <M> of a Turing machine M is the 7-tuple (S, A, Ã, t, i, S_{acc}, S_{rej}), which is a finite string over a language B that contains A and is finite.
By the theorem, $B^* = B^0 \cup B^1 \cup \ldots \cup B^\infty$ is countably infinite.