Chapter 11

Unsolvable Problems

11.1. For any language L, the identity function from Σ^* to Σ^* reduces L to itself.

Suppose $f : \Sigma^* \to \Sigma^*$ and $g : \Sigma^* \to \Sigma^*$ have the property that for every x and y in Σ^*, $x \in L_1$ if and only if $f(x) \in L_2$, and $y \in L_2$ if and only if $g(y) \in L_3$. Then for any $x \in \Sigma^*$, $x \in L_1$ if and only if $g(f(x)) \in L_3$. In addition, if f and g are computable, their composition $g \circ f$ is also computable. Therefore, if f reduces L_1 to L_2 and g reduces L_2 to L_3, $g \circ f$ reduces L_1 to L_3.

We want an example of two languages L_1 and L_2 for which $L_1 \leq L_2$ but $L_2 \not\leq L_1$. A trivial example is for L_1 to be Σ^* and for L_2 to be $\{\Lambda\}$. The constant function $f : \Sigma^* \to \Sigma^*$ defined by $f(x) = \Lambda$ for every x reduces L_1 to L_2. However, for any $g : \Sigma^* \to \Sigma^*$, it can't be true that for every x, $x = \Lambda$ if and only if $g(x) \in \Sigma^*$.

11.2. (a) Given n, is evenly divisible by 10? (Reason: for any n, n is divisible by 2 if and only if $5n$ is divisible by 10.)

(b) One possibility is

$$g(n) = \begin{cases} k/5 & \text{if } k \text{ is divisible by 5} \\ 1 & \text{otherwise} \end{cases}$$

If n is divisible by 10, then $g(n) = k/5$, which is divisible by 2. If n is not divisible by 10, then either n is divisible by 5 and not by 2, so that $g(n)$ is not divisible by 2, or n is not divisible by 5, which also implies that $g(n)$ is not divisible by 2.

11.3. Let $f : \Sigma^* \to \Sigma^*$ be a function that reduces L_1 to L_2, let T_f be a TM computing f, and let T_2 be a TM accepting L_2. Then the composite TM $T_f T_2$ accepts L_1, because if the input string x is in L_1, $f(x) \in L_2$, so that T_2 accepts $f(x)$; and if $x \notin L_1$, then $f(x) \notin L_2$, so that T_2 does not accept $f(x)$.

11.4. Let x_0 be a string in L, and let y_0 be a string in L'. For any recursive language L_1, define $f : \Sigma^* \to \Sigma^*$ by $f(x) = x_0$ if $x \in L_1$ and $f(x) = y_0$ if $x \notin L_1$. Obviously, for any x, $x \in L_1$ if and only if $f(x) \in L$. Furthermore, f is computable, because L_1 is recursive.

11.5. (See the solution to Exercise 10.8.) Choose some algorithmic way to enumerate the 4-tuples (x, y, z, n) in $\mathcal{N} \times \mathcal{N} \times \mathcal{N} \times \mathcal{N}$ for which $n \geq 3$. It is then straightforward to build a TM that tests 4-tuples in this order, and halts if and only if it finds one satisfying $x^n + y^n = z^n$. Therefore, knowing whether this TM ever halts on the empty input is equivalent to knowing whether Fermat's last theorem is false.

11.6. Suppose L is recursively enumerable, and let T be a TM accepting L. Define $f : \Sigma^* \to \Sigma^*$ by $f(x) = e(T)e(x)$. Then if $x \in L$, T accepts x, and thus $f(x) \in Acc$. If $x \notin L$, then T doesn't accept x, and $f(x) \notin Acc$. The function f is computable, since

computing $f(x)$ simply amounts to encoding the TM T and the string x.

11.7. If the language L is accepted by a TM T, and the question Given x, does T accept x? is solvable, then L is recursive, since there is an algorithm to determine whether any given string is in L.

11.8. An instance of **Accepts** is a pair (T, y), where T is a TM and y is a string. An instance of **Accepts**-x is simply a TM. Consider the algorithm that takes a pair (T, y) and constructs a TM T' to work as follows. T' starts by comparing its input to the string x. If the input string is not x, T' simply executes T on the input. If the input is x, then T' replaces it by y and then executes T on y. Then T' accepts x if and only if T accepts y. Therefore, the algorithm reduces **Accepts** to **Accepts**-x.

11.9. In both cases, an instance of the problem is a TM. We need an algorithm that takes an arbitrary TM T and constructs another TM T' having the property that T accepts Λ if and only if $L(T') = \{\Lambda\}$. We may obtain T' by letting it execute T if the input is Λ and immediately rejecting every other input. If T accepts Λ, then T' accepts Λ and rejects every other string, so that $L(T') = \{\Lambda\}$; if T does not accept Λ, then T' still rejects every nonnull input, but now it fails to accept Λ as well, so that $L(T') = \emptyset \neq \{\Lambda\}$.

11.10. (a) We may take $C = A \cup B$ and $D = A \cap B$.

(b) Given two TMs T_1 and T_2, we may construct T_1' and T_2' so that $L(T_1') = L(T_1) \cup L(T_2)$ and $L(T_2') = L(T_1) \cap L(T_2)$ (see the proof of Theorem 10.3). Then $L(T_1) = L(T_2)$ if and only if $L(T_1') \subseteq L(T_2')$. Therefore, the construction is a reduction from **Equivalent** to **Subset**.

11.11. (a) Let $C = A \cap B$ and $D = A$. Then $A \subseteq B$ if and only if $C = D$.

(b) Given T_1 and T_2, let T_1' be a TM accepting $L(T_1) \cap L(T_2)$ and let $T_2' = T_2$. Then the algorithm that constructs T_1' and T_2' from T_1 and T_2 determines a reduction from **Subset** to **Equivalent**.

11.12. (a) This problem is solvable. As T proceeds in its computation, one of the following must eventually occur: i) T changes state; ii) T crashes without having changed state; iii) T moves its tape head to the right, without having changed state; iv) without having changed state, T writes a symbol on square 0 that it has previously written. In the first two cases we have the answer to the question. In either of the last two cases, T must be in an infinite loop, and thus we may answer the question negatively.

(b) To show this problem is unsolvable, we reduce **Accepts**(Λ) to it. Given a TM T, an instance of **Accepts**(Λ), construct a pair (T_1, q), an instance of this problem, as follows. T_1 has all the states T does as well as one additional one, called q. T_1 works by making exactly the same moves as T, except that if T ever halts, T_1 moves instead to state q, then halts on the next move. Clearly T accepts the string Λ if and only if T_1 eventually enters state q.

(c) Exactly the same construction as in (b) shows that **AcceptsSomething** is reducible to this problem; therefore, this problem is unsolvable.

(d) To show that this problem is unsolvable, we reduce **Accepts**(Λ) to it. Given T, construct T_1 so that T_1 copies T but also keeps track of whether it's made an even number of moves or an odd number. (This is easy to do. Instead of using the states p of T, just use pairs $(p, 0)$ and $(p, 1)$ for the states of T_1. A move by T from p to q corresponds to moves by T_1 from $(p, 0)$ to $(q, 1)$ and from $(p, 1)$ to $(q, 0)$.) The other difference between T and T_1 is that if T accepts after an odd number of moves, then T_1 makes an additional move before accepting. The result is that T_1 accepts exactly the same strings as T, but accepts only after an even number of moves. Therefore, T accepts Λ if and only if T_1 accepts Λ after an even number of moves.

(e) Unsolvable. The same construction as in (d) reduces **AcceptsSomething** to this problem.

(f) Given a TM T, construct a new TM T_1 that accepts the same language but never rejects (see Exercise 9.11). Then for any input x, T fails to accept x if and only if T_1 loops forever on x. This means that the problem P: Given a pair (T, x), does T fail to accept x? can be reduced to P_1: Given (T, x), does T loop forever on x? Since P is unsolvable (it is the complementary problem to **Accepts**), P_1 is also.

(g) The same construction as in (f) reduces the problem Given T, is there a string T fails to accept? to this one. Since the first problem is unsolvable, so is this one.

(h) Given a TM T, it is possible to construct another one, T', having the property that for any input string x, T accepts x if and only if T' rejects x. (The construction is essentially to interchange the states h_a and h_r.) Therefore, the problem **Accepts** can be reduced to this problem, which implies that this problem is unsolvable.

(i) The construction in (h) reduces the problem **AcceptsSomething** to this one, which implies that this one is unsolvable.

(j) and (k) are both solvable. The reason is that within ten moves, a TM can't move its tape head any farther right than square 10. Therefore, if necessary we can look at the first 10 moves T makes for every possible input string of length 10 or less. At the end of this process, we will know whether T halts within 10 moves on every string, and we will know whether there are any strings for which T halts within 10 moves.

(l) Consider a specific language that is neither $\emptyset$ nor Σ^*, say $\{\Lambda\}$. Given a TM T, construct two TMs T_1 and T_2 by letting T_1 be T and T_2 be any TM that accepts precisely the language $\{\Lambda\}$. Then $L(T_1) \subseteq L(T_2)$ or $L(T_2) \subseteq L(T_1)$ if and only if $L(T) \subseteq \{\Lambda\}$ or $\{\Lambda\} \subseteq L(T)$. Therefore, the decision problem P: Given T, does $L(T)$ have the property that $L(T) \subseteq \{\Lambda\}$ or $\{\Lambda\} \subseteq L(T)$? can be reduced to this one. However, this property is a nontrivial property of recursively enumerable languages, so that P is unsolvable by Rice's Theorem. Therefore, the given problem is unsolvable.

11.14. There is an algorithm to take an arbitrary TM T and find a grammar generating the language $L(T)$. Therefore, to each of these problems, the corresponding problem involving TMs can be reduced. It follows that all four problems are unsolvable.

11.15. Suppose T has n nonhalting states. Within n moves, T will have halted or entered

some nonhalting state q for the second time. If T has not written a nonblank symbol by that time, it never will—in the second case because it's in an infinite loop.

11.16. The construction reduces a problem to an unsolvable problem, which doesn't prove anything.

11.17.

$\#$	$q_0\Delta$	a	b	$\#$	Δ	q_1a	b	$\#$	Δ	a	q_1b	$\#$
$\#q_0\Delta ab\#$	Δq_1	a	b	$\#$	Δ	aq_1	b	$\#$	Δ	a	bq_1	$\#$

Δ	a	$bq_1\#$	Δ	a	q_2b	Δ	$\#$	Δ	$ah_a\Delta$	Δ	$\#$	$\Delta h_a\Delta$	$\#$	$h_a\#\#$
Δ	a	$q_2b\Delta\#$	Δ	a	$h_a\Delta$	Δ	$\#$	Δ	h_a	Δ	$\#$	h_a	$\#$	$\#$

11.18. (a) Any partial solution must begin with domino 1. The next two dominoes are then determined, and both must be domino 2. At that point, the only domino that can follow is domino 1, but it is then impossible to follow it with anything.

(b)

1	001	01
10	0	101

11.19. We can construct a reduction from the general correspondence problem to the restricted problem in which the alphabet has only two symbols, by simply encoding more general symbols by strings of 0's and 1's. This can be done so that any string of 0's and 1's represents at most one string of the original symbols. Then there will be a solution sequence for the original instance if and only if there is a solution sequence for the instance of the restricted problem.

11.20. Given an instance (α_1, β_1), (α_2, β_2), $\ldots$, (α_n, β_n) of **PCP** in which $\alpha_i, \beta_i \in \{a\}^*$ for each i, let $d_i = |\alpha_i| - |\beta_i|$. If $d_i = 0$ for some i, the instance is obviously a yes-instance. If either $d_i > 0$ for every i or $d_i < 0$ for every i, the instance is obviously a no-instance. Finally, if $d_i = p > 0$ and $d_j = -q < 0$ for some i and j, then it is a yes-instance, because $\alpha_i^q \alpha_j^p = \beta_i^q \beta_j^p$.

11.21. (a) It is easy to show that the problem **CFGGeneratesAll** is reducible to this one.

(b) **CFGGeneratesAll** is also reducible to this one. Given a CFG G, let G_1 be a CFG generating Σ^* and $G_2 = G$. Then $L(G) = \Sigma^*$ if and only if $L(G_1) \subseteq L(G_2)$.

(c) Since Σ^* is regular, **CFGGeneratesAll** is also reducible to this problem.

11.23. We can construct the function $g : \Sigma_1^* \to \Sigma_2^*$ as follows. If $x \in \Sigma_1^*$ represents an instance of P_1, say I, then $g(x) = e_2(f(I))$, and otherwise $g(x) = z$, where z is a string in Σ_2^* that does not represent an instance of P_2. Then if $x \in Y(P_1)$, $x = e_1(I)$ for some yes-instance I of P_1; then $f(I)$ is a yes-instance of P_2, and $e_2(f(I)) \in Y(P_2)$. If $x \notin Y(P_1)$, then either x is not an instance of P_1 or $x = e_1(I)$ for some no-instance I of P_1. In either case, $g(x) \notin Y(P_2)$.

Note that there is one potential problem with this: There may be strings in Σ_1^* that are not instances of P_1 but no strings in Σ_2^* that are not instances of P_2. The statement of the

exercise should therefore include the assumption that not every string in Σ_2^* represents an instance of P_2.

11.24. Let y_0 be a fixed string encoding a no-instance of P_2, and consider the function t' defined by $t'(x) = t(x)$ if $t(x)$ is an instance of P_2, and $t'(x) = y_0$ otherwise. Since the function t reduces P_1 to P_2, clearly t' takes yes-instances to yes-instances. In addition, if x represents a no-instance of P_1, then $t'(x)$ represents a no-instance of P_2: If $t(x)$ already represents a no-instance of P_2, $t'(x) = t(x)$, and otherwise, $t'(x) = y_0$. t' is computable, because t is and because it is possible to determine algorithmically whether a string represents an instance of P_2.

11.25. According to the assumption, if I is an instance of P_1, $t(e_1(I))$ represents an instance J of P_2. It is easy to check that the function f that assigns to each I the corresponding J is in fact a reduction of P_1 to P_2.

11.26. (a) We can define $f : \{0,1\}^* \to \{0,1\}^*$ as follows. For a string x not of the form $e(T)e(w)$, $f(x) = \Lambda$; if $x = e(T)e(w)$, $f(x) = e(T')$, where the TM T' works by substituting the string w for its input and then executing T on w. If $x \in Acc$ (i.e., $x = e(T)e(w)$ and T accepts w), then $f(x) = e(T')$, where T' accepts every string, so that $f(x) \in AE$. If $x \notin Acc$, then either $f(x) = e(T')$ for some T' not accepting everything (in fact, not accepting anything), or $f(x) = \Lambda$; in either case, $f(x) \notin AE$.

(b) Any function that determines a reduction from Acc to AE also determines a reduction from Acc' to AE'.

(c) The language Acc is recursively enumerable. We can construct a TM T to accept Acc as follows. T tests its input string to see if it is of the form $e(T)e(w)$. If not, it rejects; if so, it then executes a universal TM on the original input. The result is that T accepts the input precisely if it is an element of Acc. Now, since we know that the decision problem **Accepts** is unsolvable, Acc cannot be recursive. Therefore, by Theorem 10.5, Acc' cannot be recursively enumerable. Therefore, by Exercise 11.3, AE' is not recursively enumerable.

(d) Let T_0 be a TM that immediately accepts, no matter what the input. To complete the definition of f we must say only what $f(x)$ is for a string x not of the form $e(T)e(z)$. Since such an x is an element of Acc', we want $f(x)$ to be an element of AE; for this reason, we define $f(x)$ to be $e(T_0)$ in this case.

Now if $x = e(T)e(z)$, the string $f(x)$ is $e(S_{T,z})$. In the case where T accepts z, say in n moves, the computation performed by $S_{T,z}$ on an input string of length $\geq n$ causes it to enter an infinite loop, because by the time it finishes simulating n moves of T on input z it will discover that T accepts z. Therefore, in this case, the TM $S_{T,z}$ does not accept every string, which implies that $f(x) \notin AE$. In the other case, where T does not accept z, $S_{T,z}$ does accept every input, because no matter how long its input is, simulating that number of moves of T on input z will not cause T to accept. The conclusion is that f does determine a reduction from Acc' to AE.

(e) If AE were recursively enumerable, it would follow from the previous part and Exercise 11.3 that Acc' would be recursively enumerable. But we have seen in part (c) that it is not.

11.28. Suppose L is a recursively enumerable language that is not recursive (for example, the language Acc introduced in Exercise 11.26). Then by Theorem 10.5, L' is not recursively enumerable. On the one hand, $L' \not\leq L$, by the result in Exercise 11.3. On the other hand, if $L \leq L'$, then the same function that reduces L to L' also reduces L' to $(L')' = L$; therefore $L \not\leq L'$.

11.29. (a) Given a TM T, we construct a TM T' as follows. T' first compares its input to y; if the input is different from y, it simply executes T on its input, and if the input is y, T' replaces it by x before executing T. Then T accepts x if and only if T' accepts y. Therefore, this construction defines a reduction from $P_{\{x\}}$ to $P_{\{y\}}$.

(b) To get a reduction from $P_{\{x\}}$ to $P_{\{y,z\}}$, we can use the same construction as in (a), except that T' replaces both the input y and the input z by x. It follows that T accepts x if and only if T' accepts both y and z.

(c) For this part the TM T' is constructed from T as follows. If the input is anything other than z, T' executes T. If the input is z, T' first replaces z by x and executes T on x. If (and only if) that computation would cause T to accept x, T' then erases its tape, writes y on the tape, and executes T on the input y, accepting precisely if T would accept y. Then T accepts both x and y if and only if T' accepts z. This construction provides the reduction from $P_{\{x,y\}}$ to $P_{\{z\}}$.

(d) If S is any nonempty finite set, the argument in (c) shows that $P_S \leq P_{\{\Lambda\}}$. If U is any nonempty finite set, the argument in (b) shows that $P_{\{\Lambda\}} \leq P_U$. It follows that for any two finite nonempty sets S and U, $P_S \leq P_U$.

11.30. (a) We can construct T' from T as in (a) of the previous exercise, except that here T' replaces the input y by the input x and vice versa before executing T. This guarantees that T accepts x and nothing else, if and only if T' accepts y and nothing else.

In (b), T' replaces both y and z by x. As in the previous exercise, this guarantees that if T accepts x, then T' accepts y and z, and conversely. Now, since on input y and z, T' does not actually execute T on those strings, we must be careful to guarantee that if T' accepts nothing besides y and z, then T accepts nothing but x. (In particular, we must make sure it doesn't accept y or z.)

Let us assume that $x \notin \{y, z\}$ and that in lexicographic order, x precedes y and y precedes z. The other cases can be handled similarly. We wish to fix it so that the strings on which T' actually executes T, if the input of T' is neither y nor z, include all the strings other than x. T' can replace input x by y before executing T. In addition, for every string that comes after z in lexicographic order, T' replaces that input by the string that immediately precedes it.

Now, if T accepts x and nothing else, then T' accepts y and z and nothing else. (It doesn't accept x, because input x has been replaced by y, which is not one of the strings T accepts.) Conversely, if T' accepts y and z and nothing else, then T accepts x. Furthermore, it doesn't accept any other strings, because for any $w \neq x$, some input other than y or z would cause T' to process w by running T on it. This gives us the reduction we need.

(c) We assume that $z \notin \{x, y\}$. As in (c) of the previous exercise, we have T' replace input z by x, run T on x, and if it accepts, run it on input y. This will guarantee that if T

accepts x and y, then T' accepts z, and conversely. Now we must see to it that the strings on which T can be executed by T', if the input to T' is not z, include z but neither x nor y. We can accomplish this by simply having T' replace both input x and input y by z before running T. It follows that T accepts x and y and nothing else, if and only if T' accepts z and nothing else.

(d) As in the previous exercise, the techniques in (b) and (c) can be easily modified to show that for any nonempty finite sets S and U, $P_S \leq P_{\{\Lambda\}} \leq P_U$.

11.31. (a) To show this problem is unsolvable, we can reduce **Accepts-Λ** to it. Given a TM T, an instance of **Accepts-Λ**, we construct another TM T_1 as follows. T_1 has all of T's states, as well as one additional state q; it has all of T's tape symbols, and one additional one, \$. The transitions of T_1 are the same as those of T, with these additions and modifications. For any accepting move $\delta(p, a) = (h_a, b, D)$, T_1 has instead the move $\delta_{T_1}(p, a) = (q_0, \$, \mathrm{S})$, where q_0 is the initial state of T. If the nonhalting states of T are enumerated $q_0, q_1, \ldots, q_n$, then T_1 has the additional transitions $\delta_T(q_i, \$) = (q_{i+1}, \$, \mathrm{S})$ (for each i with $0 \leq i < n$); finally, $\delta_T(q_n, \$) = (q, \$, \mathrm{S})$ and $\delta_T(q, \$) = (h_a, \$, \mathrm{S})$.

To summarize: for any move that would cause T to accept, T_1 instead moves to q_0 and places \$ on the tape; thereafter, the \$ causes T_1 to cycle through all its nonhalting states, q being the last, before accepting. On the one hand, if T accepts Λ, then some move causes T to accept, and therefore, T_1 enters all its nonhalting states when started with a blank tape. Conversely, if T doesn't accept Λ, then since T never executes an accepting move after starting with a blank tape, T_1 will never enter the state q, and so does not enter all its nonhalting states.

(b) The construction in (a) reduces the problem **AcceptsSomething** to this problem. It follows that this problem is unsolvable.

11.32. We sketch the way an LBA might operate if it is to accept strings representing solutions to the given correspondence system. It starts by creating a second "track" to the tape, which is initially blank. Next, it nondeterministically attempts to decompose the string in the first track (i.e., the input string) into consecutive pieces, each of which is an α_i. The end of each α_i is marked somehow. Assuming the LBA is able to do this successfully, it makes another pass through the string, constructing in the second track of the tape a string of β_i's corresponding to the α_i's in the first track. (This step may require nondeterminism also, if there are strings α_i corresponding to two different β_i's.) Finally, if the resulting string in the second track is exactly the right length, the machine compares the first and second tracks and accepts if the two strings are equal.

11.33. (a) One way to enumerate the CSGs is the following. First, list all CSGs for which the only variable is A_1 and every production $\alpha \to \beta$ has $|\alpha|, |\beta| \leq 1$; there are only finitely many. Next, list those not already listed for which the only variables are A_1 and A_2 and every production $\alpha \to \beta$ has $|\alpha|, |\beta| \leq 2$. If we continue in this way, every CSG satisfying our assumptions will eventually be listed.

(b) Each language $L(G_i)$ is recursive, by Theorem 10.12. Therefore, given $x \in \{a, b\}^*$, we can determine whether $x \in L$ by finding the i for which $x = x_i$ and then testing x for

membership in $L(G_i)$. Therefore, L is recursive. If L is context-sensitive, then L is the same as $L(G_i)$ for some i. In this case, consider whether the corresponding x_i can be an element of L. If it is, then by definition of L, $x \notin L(G_i) = L$. If it is not, however, then $x_i \notin L(G_i)$, and so $x \in L$ by definition of L. This contradiction proves that L cannot be context-sensitive.

11.34. This problem is solvable. A decision algorithm is the following. First test x for membership in $L(G)$. If $x \notin L(G)$, then $L(G) \neq \{x\}$. If $x \in L(G)$, then construct a PDA M accepting $L(G)$, using Section 7.4. Since $L_1 = \{z \in \Sigma^* \mid z \neq x\}$ is regular, the proof of Theorem 8.4 provides an algorithm for constructing another PDA M_1 to accept $L \cap L_1$. Section 7.5 describes an algorithm to produce a CFG G_1 generating $L(M_1)$. In Section 8.3 there is a decision algorithm to decide whether $L(G_1) = \emptyset$. If $L(G_1) = \emptyset$, then $L(G) = \{x\}$; otherwise, $L(G) \neq \{x\}$.

11.35. Using the same technique as in Exercise 11.34, we can construct a CFG generating $L(G) - R$, and we can determine whether this language is nonempty. Since $L(G) \subseteq R$ if and only if $L(G) - R = \emptyset$, this problem is solvable.

11.37. Let I be an instance of **PCP**. As suggested, let G_α and G_β be the CFGs constructed from I. Although the complement of a CFL is not in general a CFL, in this case we can actually construct CFGs G'_α and G'_β generating the complements of $L(G_\alpha)$ and $L(G_\beta)$, respectively. First we can construct a deterministic PDA M_α to accept $L(G_\alpha)$. It works by initially reading and pushing onto its stack all input symbols that are elements of Σ. If and when one of the c_i's is read for the first time, the PDA crashes unless it is able to pop from its stack the symbols of the corresponding α_i in reverse. Thereafter, the only legal inputs are c_i's, and the PDA continues matching them with the reverse of the strings on the stack. If this continues successfully until M_α finally pops from the stack the first α_i put there, so that the stack is empty except for Z_0, M_α accepts. The PDA M_β can be constructed the same way.

Next we construct M'_α and M'_β from M_α and M_β by making the accepting states nonaccepting and vice-versa. In this case (although not in general), the resulting PDAs accept the complements of the original languages. The CFGs G'_α and G'_β are then constructed from the PDAs M'_α and M'_β.

Finally, let G' be a CFG generating the language $L(G'_\alpha) \cup L(G'_\beta) = L(G_\alpha)' \cup L(G_\beta)'$. The instance I is a yes-instance of **PCP** if and only if $L(G_\alpha) \cap L(G_\beta)$ is nonempty. This is true if and only if the complement of this language is different from Σ^*. But the complement of this language is $L(G')$. We have therefore established a reduction of **PCP** to the given problem.