We can have our program do any particular thing we want, e.g., halt or print `hello, world`, when and if it finds a solution. Otherwise, the program will never perform that particular action. Thus, it is undecidable whether a program prints `hello, world`, whether it halts, whether it calls a particular function, rings the console bell, or makes any other nontrivial action. In fact, there is an analog of Rice's Theorem for programs: any nontrivial property that involves what the program does (rather than a lexical or syntactic property of the program itself) must be undecidable.

9.5.2 Undecidability of Ambiguity for CFG's

Programs are sufficiently like Turing machines that the observations of Section 9.5.1 are unsurprising. Now, we shall see how to reduce PCP to a problem that looks nothing like a question about computers: the question of whether a given context-free grammar is ambiguous.

The key idea is to consider strings that represent a list of indexes (integers), in reverse, and the corresponding strings according to one of the lists of a PCP instance. These strings can be generated by a grammar. The similar set of strings for the other list in the PCP instance can also be generated by a grammar. If we take the union of these grammars in the obvious way, then there is a string generated through the productions of each original grammar if and only if there is a solution to this PCP instance. Thus, there is a solution if and only if there is ambiguity in the grammar for the union.

Let us now make these ideas more precise. Let the PCP instance consist of lists $A = w_1, w_2, \ldots, w_k$ and $B = x_1, x_2, \ldots, x_k$. For list A we shall construct a CFG with A as the only variable. The terminals are all the symbols of the alphabet Σ used for this PCP instance, plus a distinct set of *index symbols* $a_1, a_2, \ldots, a_k$ that represent the choices of pairs of strings in a solution to the PCP instance. That is, the index symbol a_i represents the choice of w_i from the A list or x_i from the B list. The productions for the CFG for the A list are:

$$\begin{aligned} A \quad \rightarrow \quad & w_1 A a_1 \mid w_2 A a_2 \mid \cdots \mid w_k A a_k \mid \\ & w_1 a_1 \mid w_2 a_2 \mid \cdots \mid w_k a_k \end{aligned}$$

We shall call this grammar G_A and its language L_A. In the future, we shall refer to a language like L_A as *the language for the list* A.

Notice that the terminal strings derived by G_A are all those of the form $w_{i_1} w_{i_2} \cdots w_{i_m} a_{i_m} \cdots a_{i_2} a_{i_1}$ for some $m \geq 1$ and list of integers $i_1, i_2, \ldots, i_m$; each integer is in the range 1 to k. The sentential forms of G_A all have a single A between the strings (the w's) and the index symbols (the a's), until we use one of the last group of k productions, none of which have an A in the body. Thus, parse trees look like the one suggested in Fig. 9.16.

Observe also that any terminal string derivable from A in G_A has a unique derivation. The index symbols at the end of the string determine uniquely which production must be used at each step. That is, only two production bodies end with a given index symbol a_i: $A \rightarrow w_i A a_i$ and $A \rightarrow w_i a_i$. We must

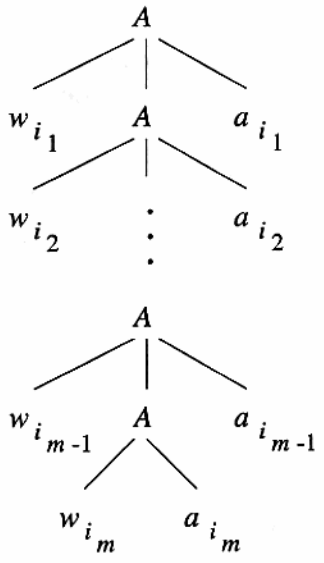

Figure 9.16: The form of parse trees in the grammar G_A

use the first of these if the derivation step is not the last, and we must use the second production if it is the last step.

Now, let us consider the other part of the given PCP instance, the list $B = x_1, x_2, \ldots, x_k$. For this list we develop another grammar G_B:

$$\begin{aligned} B \quad \rightarrow \quad & x_1 B a_1 \mid x_2 B a_2 \mid \cdots \mid x_k B a_k \mid \\ & x_1 a_1 \mid x_2 a_2 \mid \cdots \mid x_k a_k \end{aligned}$$

The language of this grammar will be referred to as L_B. The same observations that we made for G_A apply also to G_B. In particular, a terminal string in L_B has a unique derivation, which can be determined by the index symbols in the tail of the string.

Finally, we combine the languages and grammars of the two lists to form a grammar G_{AB} for the entire PCP instance. G_{AB} consists of:

1. Variables A, B, and S; the latter is the start symbol.
2. Productions $S \rightarrow A \mid B$.
3. All the productions of G_A.
4. All the productions of G_B.

We claim that G_{AB} is ambiguous if and only if the instance (A, B) of PCP has a solution; that argument is the core of the next theorem.

Theorem 9.20: It is undecidable whether a CFG is ambiguous.

PROOF: We have already given most of the reduction of PCP to the question of whether a CFG is ambiguous; that reduction proves the problem of CFG ambiguity to be undecidable, since PCP is undecidable. We have only to show that the above construction is correct; that is:

- G_{AB} is ambiguous if and only if instance (A, B) of PCP has a solution.

(If) Suppose $i_1, i_2, \ldots, i_m$ is a solution to this instance of PCP. Consider the two derivations in G_{AB}:

$$S \Rightarrow A \Rightarrow w_{i_1}Aa_{i_1} \Rightarrow w_{i_1}w_{i_2}Aa_{i_2}a_{i_1} \Rightarrow \cdots \Rightarrow$$
$$w_{i_1}w_{i_2}\cdots w_{i_{m-1}}Aa_{i_{m-1}}\cdots a_{i_2}a_{i_1} \Rightarrow w_{i_1}w_{i_2}\cdots w_{i_m}a_{i_m}\cdots a_{i_2}a_{i_1}$$

$$S \Rightarrow B \Rightarrow x_{i_1}Ba_{i_1} \Rightarrow x_{i_1}x_{i_2}Ba_{i_2}a_{i_1} \Rightarrow \cdots \Rightarrow$$
$$x_{i_1}x_{i_2}\cdots x_{i_{m-1}}Ba_{i_{m-1}}\cdots a_{i_2}a_{i_1} \Rightarrow x_{i_1}x_{i_2}\cdots x_{i_m}a_{i_m}\cdots a_{i_2}a_{i_1}$$

Since $i_1, i_2, \ldots, i_m$ is a solution, we know that $w_{i_1}w_{i_2}\cdots w_{i_m} = x_{i_1}x_{i_2}\cdots x_{i_m}$. Thus, these two derivations are derivations of the same terminal string. Since the derivations themselves are clearly two distinct, leftmost derivations of the same terminal string, we conclude that G_{AB} is ambiguous.

(Only-if) We already observed that a given terminal string cannot have more than one derivation in G_A and not more than one in G_B. So the only way that a terminal string could have two leftmost derivations in G_{AB} is if one of them begins $S \Rightarrow A$ and continues with a derivation in G_A, while the other begins $S \Rightarrow B$ and continues with a derivation of the same string in G_B.

The string with two derivations has a tail of indexes $a_{i_m}\cdots a_{i_2}a_{i_1}$, for some $m \geq 1$. This tail must be a solution to the PCP instance, because what precedes the tail in the string with two derivations is both $w_{i_1}w_{i_2}\cdots w_{i_m}$ and $x_{i_1}x_{i_2}\cdots x_{i_m}$. □

9.5.3 The Complement of a List Language

Having context-free languages like L_A for the list A lets us show a number of problems about CFL's to be undecidable. More undecidability facts for CFL's can be obtained by considering the complement language $\overline{L_A}$. Notice that the language $\overline{L_A}$ consists of all strings over the alphabet $\Sigma \cup \{a_1, a_2, \ldots, a_k\}$ that are not in L_A, where Σ is the alphabet of some instance of PCP, and the a_i's are distinct symbols representing the indexes of pairs in that PCP instance.

The interesting members of $\overline{L_A}$ are those strings consisting of a prefix in Σ^* that is the concatenation of some strings from the A list, followed by a suffix of index symbols that does *not* match the strings from A. However, there are also many strings in $\overline{L_A}$ that are simply of the wrong form: they are not in the language of regular expression $\Sigma^*(a_1 + a_2 + \cdots + a_k)^*$.

We claim that $\overline{L_A}$ is a CFL. Unlike L_A, it is not very easy to design a grammar for $\overline{L_A}$, but we can design a PDA, in fact a deterministic PDA, for $\overline{L_A}$. The construction is in the next theorem.

Theorem 9.21: If L_A is the language for list A, then $\overline{L_A}$ is a context-free language.

PROOF: Let Σ be the alphabet of the strings on list $A = w_1, w_2, \ldots, w_k$, and let I be the set of index symbols: $I = \{a_1, a_2, \ldots, a_k\}$. The DPDA P we design to accept $\overline{L_A}$ works as follows.

1. As long as P sees symbols in Σ, it stores them on its stack. Since all strings in Σ^* are in $\overline{L_A}$, P accepts as it goes.
2. As soon as a P sees an index symbol in I, say a_i, it pops its stack to see if the top symbols form w_i^R, that is, the reverse of the corresponding string.
   (a) If not, then the input seen so far, and any continuation of this input is in $\overline{L_A}$. Thus, P goes to an accepting state in which it consumes all future inputs without changing its stack.
   (b) If w_i^R was popped from the stack, but the bottom-of-stack marker is not yet exposed on the stack, then P accepts, but remembers, in its state that it is looking for symbols in I only, and may yet see a string in L_A (which P will *not* accept). P repeats step (2) as long as the question of whether the input is in L_A is unresolved.
   (c) If w_i^R was popped from the stack, and the bottom-of-stack marker is exposed, then P has seen an input in L_A. P does not accept this input. However, since any input continuation cannot be in L_A, P goes to a state where it accepts all future inputs, leaving the stack unchanged.
3. If, after seeing one or more symbols of I, P sees another symbol of Σ, then the input is not of the correct form to be in L_A. Thus, P goes to a state in which it accepts this and all future inputs, without changing its stack.

□

We can use L_A, L_B and their complements in various ways to show undecidability results about context-free languages. The next theorem summarizes some of these facts.

Theorem 9.22: Let G_1 and G_2 be context-free grammars, and let R be a regular expression. Then the following are undecidable:

a) Is $L(G_1) \cap L(G_2) = \emptyset$?

b) Is $L(G_1) = L(G_2)$?

c) Is $L(G_1) = L(R)$?

d) Is $L(G_1) = T^*$ for some alphabet T?

e) Is $L(G_1) \subseteq L(G_2)$?

f) Is $L(R) \subseteq L(G_1)$?

PROOF: Each of the proofs is a reduction from PCP. We show how to take an instance (A, B) of PCP and convert it to a question about CFG's and/or regular expressions that has answer "yes" if and only if the instance of PCP has a solution. In some cases, we reduce PCP to the question as stated in the theorem; in other cases we reduce it to the complement. It doesn't matter, since if we show the complement of a problem to be undecidable, it is not possible that the problem itself is decidable, since the recursive languages are closed under complementation (Theorem 9.3).

We shall refer to the alphabet of the strings for this instance as Σ and the alphabet of index symbols as I. Our reductions depend on the fact that L_A, L_B, $\overline{L_A}$, and $\overline{L_B}$ all have CFG's. We construct these CFG's either directly, as in Section 9.5.2, or by the construction of a PDA for the complement languages given in Theorem 9.21 coupled with the conversion from a PDA to a CFG by Theorem 6.14.

a) Let $L(G_1) = L_A$ and $L(G_2) = L_B$. Then $L(G_1) \cap L(G_2)$ is the set of solutions to this instance of PCP. The intersection is empty if and only if there is no solution. Note that, technically, we have reduced PCP to the language of pairs of CFG's whose intersection is nonempty; i.e., we have shown the problem "is the intersection of two CFG's nonempty" to be undecidable. However, as mentioned in the introduction to the proof, showing the complement of a problem to be undecidable is tantamount to showing the problem itself undecidable.

b) Since CFG's are closed under union, we can construct a CFG G_1 for $\overline{L_A} \cup \overline{L_B}$. Since $(\Sigma \cup I)^*$ is a regular set, we surely may construct for it a CFG G_2. Now $\overline{L_A} \cup \overline{L_B} = \overline{L_A \cap L_B}$. Thus, $L(G_1)$ is missing only those strings that represent solutions to the instance of PCP. $L(G_2)$ is missing no strings in $(\Sigma \cup I)^*$. Thus, their languages are equal if and only if the PCP instance has no solution.

c) The argument is the same as for (b), but we let R be the regular expression $(\Sigma \cup I)^*$.

d) The argument of (c) suffices, since $\Sigma \cup I$ is the only alphabet of which $\overline{L_A} \cup \overline{L_B}$ could possibly be the closure.

e) Let G_1 be a CFG for $(\Sigma \cup I)^*$ and let G_2 be a CFG for $\overline{L_A} \cup \overline{L_B}$. Then $L(G_1) \subseteq L(G_2)$ if and only if $\overline{L_A} \cup \overline{L_B} = (\Sigma \cup I)^*$, i.e., if and only if the PCP instance has no solution.

f) The argument is the same as (e), but let R be the regular expression $(\Sigma \cup I)^*$, and let $L(G_1)$ be $\overline{L_A} \cup \overline{L_B}$.

□