Formal Languages, Automata and Codes

Oleg Gutik

Lecture 15

5.2 Parsing and Ambiguity

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

5.2 Parsing and Ambiguity

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

5.2 Parsing and Ambiguity

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

We have so far concentrated on the generative aspects of grammars. Given a grammar G, we studied the set of strings that can be derived using G. In cases of practical applications, we are also concerned with the analytical side of the grammar: Given a string w of terminals, we want to know whether or not w is in $L(G)$. If so, we may want to find a derivation of w. An algorithm that can tell us whether w is in $L(G)$ is a membership algorithm. The term parsing describes finding a sequence of productions by which a word $w \in L(G)$ is derived.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Parsing and Membership

Given a string w in $L(G)$, we can parse it in a rather obvious fashion: We systematically construct all possible (say, leftmost) derivations and see whether any of them match w. Specifically, we start at round one by looking at all productions of the form

$$S \to x,$$

finding all x that can be derived from S in one step. If none of these results in a match with w, we go to the next round, in which we apply all applicable productions to the leftmost variable of every x. This gives us a set of sentential forms, some of them possibly leading to w. On each subsequent round, we again take all leftmost variables and apply all possible productions. It may be that some of these sentential forms can be rejected on the grounds that w can never be derived from them, but in general, we shall have on each round a set of possible sentential forms. After the first round, we have sentential forms that can be derived by applying a single production, after the second round we have the sentential forms that can be derived in two steps, and so on. If $w \in L(G)$, then it must have a leftmost derivation of finite length. Thus, the method will eventually give a leftmost derivation of w.

For reference below, we will call this exhaustive search parsing or brute force parsing. It is a form of top-down parsing, which we can view as the construction of a derivation tree from the root down.

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$1.\ S \Rightarrow SS,$$
$$2.\ S \Rightarrow aSb,$$
$$3.\ S \Rightarrow bSa,$$
$$4.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$
$$S \Rightarrow SS \Rightarrow aSbS,$$
$$S \Rightarrow SS \Rightarrow bSaS,$$
$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$
$$S \Rightarrow aSb \Rightarrow aaSbb,$$
$$S \Rightarrow aSb \Rightarrow abSab,$$
$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$
$$\mathbf{2}.\ S \Rightarrow aSb,$$
$$\mathbf{3}.\ S \Rightarrow bSa,$$
$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$
$$S \Rightarrow SS \Rightarrow aSbS,$$
$$S \Rightarrow SS \Rightarrow bSaS,$$
$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$
$$S \Rightarrow aSb \Rightarrow aaSbb,$$
$$S \Rightarrow aSb \Rightarrow abSab,$$
$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7

Consider the grammar

$$S \to SS|aSb|bSa|\lambda$$

and the string $w = aabb$. Round one gives us

$$\mathbf{1}.\ S \Rightarrow SS,$$

$$\mathbf{2}.\ S \Rightarrow aSb,$$

$$\mathbf{3}.\ S \Rightarrow bSa,$$

$$\mathbf{4}.\ S \Rightarrow \lambda$$

The last two of these can be removed from further consideration for obvious reasons. Round two then yields sentential forms

$$S \Rightarrow SS \Rightarrow SSS,$$

$$S \Rightarrow SS \Rightarrow aSbS,$$

$$S \Rightarrow SS \Rightarrow bSaS,$$

$$S \Rightarrow SS \Rightarrow S,$$

which are obtained by replacing the leftmost S in sentential form 1 with all applicable substitutes. Similarly, from sentential form 2 we get the additional sentential forms

$$S \Rightarrow aSb \Rightarrow aSSb,$$

$$S \Rightarrow aSb \Rightarrow aaSbb,$$

$$S \Rightarrow aSb \Rightarrow abSab,$$

$$S \Rightarrow aSb \Rightarrow ab.$$

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \rightarrow \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \rightarrow \lambda$ as well as those of the form $A \rightarrow B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \rightarrow \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \rightarrow \lambda$ as well as those of the form $A \rightarrow B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \rightarrow \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \rightarrow \lambda$ as well as those of the form $A \rightarrow B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.7 (continuation)

Again, several of these can be removed from contention. On the next round, we find the actual target string from the sequence

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb.$$

Therefore, $aabb$ is in the language generated by the grammar under consideration.

Exhaustive search parsing has serious flaws. The most obvious one is its tediousness; it is not to be used where efficient parsing is required. But even when efficiency is a secondary issue, there is a more pertinent objection. While the method always parses a word $w \in L(G)$, it is possible that it never terminates for strings not in $L(G)$. This is certainly the case in the previous example; with $w = abb$, the method will go on producing trial sentential forms indefinitely unless we build into it some way of stopping.

The problem of nontermination of exhaustive search parsing is relatively easy to overcome if we restrict the form that the grammar can have. If we examine Example 5.7, we see that the difficulty comes from the productions $S \to \lambda$; this production can be used to decrease the length of successive sentential forms, so that we cannot tell easily when to stop. If we do not have any such productions, then we have many fewer difficulties. In fact, there are two types of productions we want to rule out, those of the form $A \to \lambda$ as well as those of the form $A \to B$. As we will see in the next lectures, this restriction does not affect the power of the resulting grammars in any significant way.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a,b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a,b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Example 5.8

The grammar

$$S \to SS|aSb|bSa|ab|ba$$

satisfies the given requirements. It generates the language in Example 5.7 without the empty string.

Given any $w \in \{a, b\}^+$, the exhaustive search parsing method will always terminate in no more than $|w|$ rounds. This is clear because the length of the sentential form grows by at least one symbol in each round. After $|w|$ rounds we have either produced a parsing or we know that $w \notin L(G)$.

The idea in this example can be generalized and made into a theorem for context-free languages in general.

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ■

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ■

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ∎

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ■

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ■

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. $\blacksquare$

Theorem 5.2

Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of the form

$$A \to \lambda,$$

or

$$A \to B.$$

where $A, B \in V$. Then the exhaustive search parsing method can be made into an algorithm that, for any $w \in \Sigma^*$, either produces a parsing of w or tells us that no parsing is possible.

Proof. For each sentential form, consider both its length and the number of terminal symbols. Each step in the derivation increases at least one of these. Since neither the length of a sentential form nor the number of terminal symbols can exceed $|w|$, a derivation cannot involve more than $2|w|$ rounds, at which time we either have a successful parsing or w cannot be generated by the grammar. ■

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

5.2 Parsing and Ambiguity

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \quad (1)$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \quad (1)$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \quad (1)$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \qquad (1)$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

5.2 Parsing and Ambiguity

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

While the exhaustive search method gives a theoretical guarantee that parsing can always be done, its practical usefulness is limited because the number of sentential forms generated by it may be excessively large. Exactly how many sentential forms are generated differs from case to case; no precise general result can be established, but we can put some rough upper bounds on it. If we restrict ourselves to leftmost derivations, we can have no more than $|P|$ sentential forms after one round, no more than $|P|^2$ sentential forms after the second round, and so on. In the proof of Theorem 5.2, we observed that parsing cannot involve more than $2|w|$ rounds; therefore, the total number of sentential forms cannot exceed

$$M = |P| + |P|^2 + \cdots + |P|^{2|w|} = O(P^{2|w|+1}). \tag{1}$$

This indicates that the work for exhaustive search parsing may grow exponentially with the length of the string, making the cost of the method prohibitive. Of course, Equation (1) is only a bound, and often the number of sentential forms is much smaller. Nevertheless, practical observation shows that exhaustive search parsing is very inefficient in most cases.

The construction of more efficient parsing methods for context-free grammars is a complicated matter that belongs to a course on compilers. We shall not pursue it here except for some isolated results.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Theorem 5.3

For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

There are several known methods to achieve this, but all of them are sufficiently complicated that we cannot even describe them without developing some additional results. In next lectures we shall take this question up again briefly. More details can be found in Harrison (1978) and Hopcroft and Ullman (1979). One reason for not pursuing this in detail is that even these algorithms are unsatisfactory. A method in which the work rises with the third power of the length of the string, while better than an exponential algorithm, is still quite inefficient, and a parser based on it would need an excessive amount of time to analyze even a moderately long program. What we would like to have is a parsing method that takes time proportional to the length of the string. We refer to such a method as a linear time parsing algorithm. We do not know any linear time parsing methods for context-free languages in general, but such algorithms can be found for restricted, but important, special cases.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \rightarrow ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \rightarrow aS|bSS|c$$

is an s-grammar. The grammar

$$S \rightarrow aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \rightarrow aS$ and $S \rightarrow aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form
$$A \to ax,$$
where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar
$$S \to aS|bSS|c$$
is an s-grammar. The grammar
$$S \to aS|bSS|aSS|c$$
is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

Definition 5.4

A context-free grammar $G = (V, T, S, P)$ is said to be a *simple grammar* or *s-grammar* if all its productions are of the form

$$A \to ax,$$

where $A \in V$, $a \in T$, $x \in V^*$, and any pair (A, a) occurs at most once in P.

Example 5.9

The grammar

$$S \to aS|bSS|c$$

is an s-grammar. The grammar

$$S \to aS|bSS|aSS|c$$

is not an s-grammar because the pair (S, a) occurs in the two productions $S \to aS$ and $S \to aSS$.

While s-grammars are quite restrictive, they are of some interest. As we will see in the next lecture, many features of common programming languages can be described by s-grammars.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

5.2 Parsing and Ambiguity

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \overset{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1A_1A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \overset{*}{\Rightarrow} a_1a_2B_1B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1A_1A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \overset{*}{\Rightarrow} a_1a_2B_1B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1A_1A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \overset{*}{\Rightarrow} a_1a_2B_1B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

5.2 Parsing and Ambiguity

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \overset{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1A_1A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1a_2B_1B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

5.2 Parsing and Ambiguity

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

5.2 Parsing and Ambiguity

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1A_1A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1a_2B_1B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

If G is an s-grammar, then any string w in $L(G)$ can be parsed with an effort proportional to $|w|$. To see this, look at the exhaustive search method and the string $w = a_1 a_2 \cdots a_n$. Since there can be at most one rule with S on the left, and starting with a_1 on the right, the derivation must begin with

$$S \Rightarrow a_1 A_1 A_2 \cdots A_m.$$

Next, we substitute for the variable A_1, but since again there is at most one choice, we must have

$$S \stackrel{*}{\Rightarrow} a_1 a_2 B_1 B_2 \cdots A_2 \cdots A_m.$$

We see from this that each step produces one terminal symbol and hence the whole process must be completed in no more than $|w|$ steps.

Ambiguity in Grammars and Languages

On the basis of our argument we can claim that given any $w \in L(G)$, exhaustive search parsing will produce a derivation tree for w. We say "a" derivation tree rather than "the" derivation tree because of the possibility that a number of different derivation trees may exist. This situation is referred to as *ambiguity*.

Definition 5.5

A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

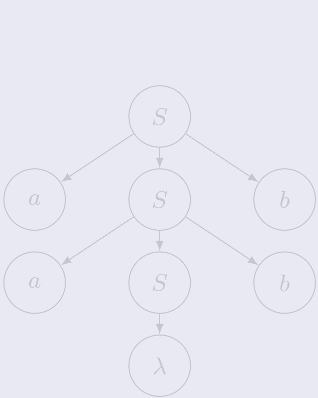
Example 5.10

The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.

Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.

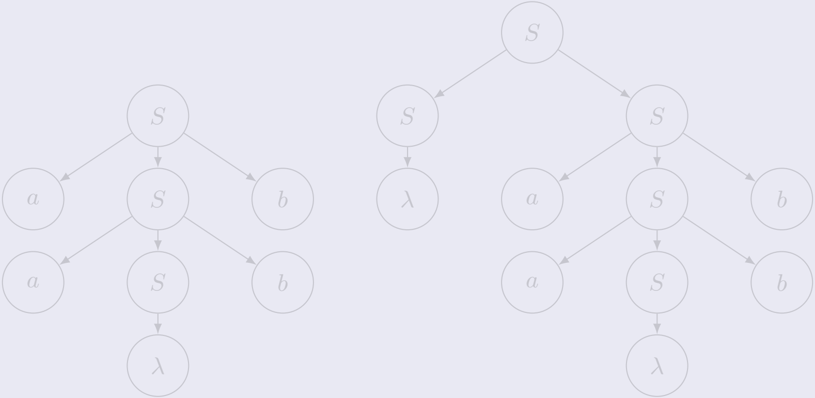


Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous.

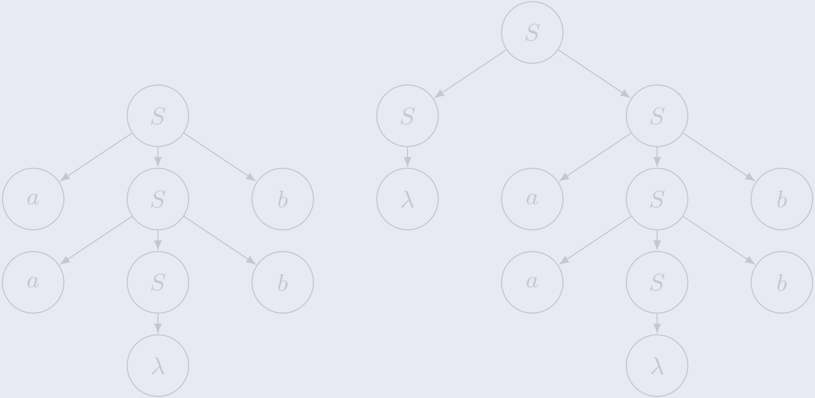
The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

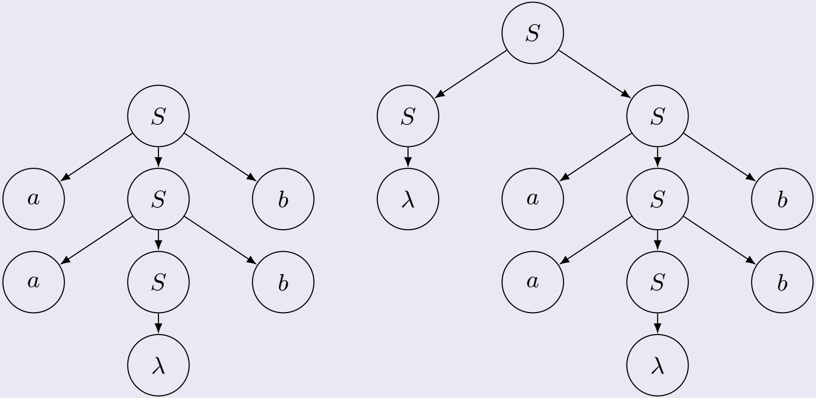
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

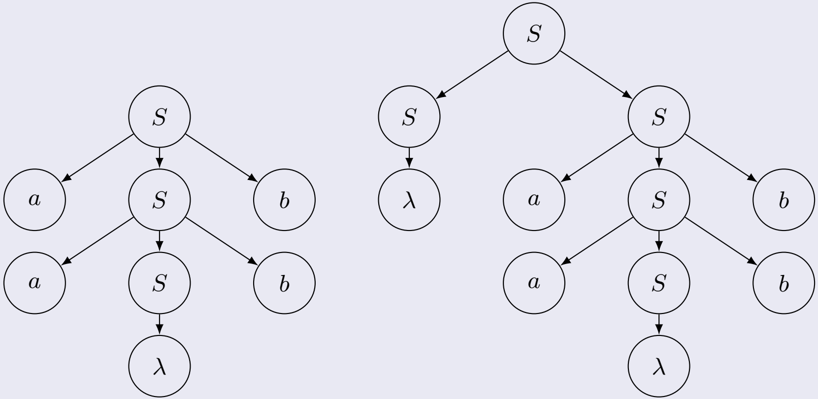
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

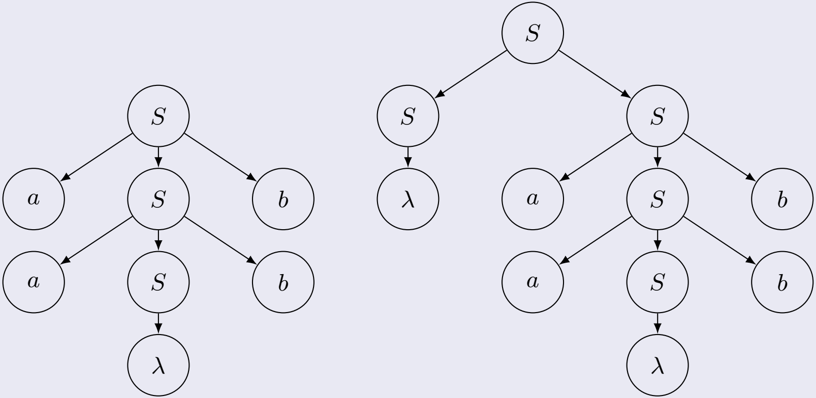
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

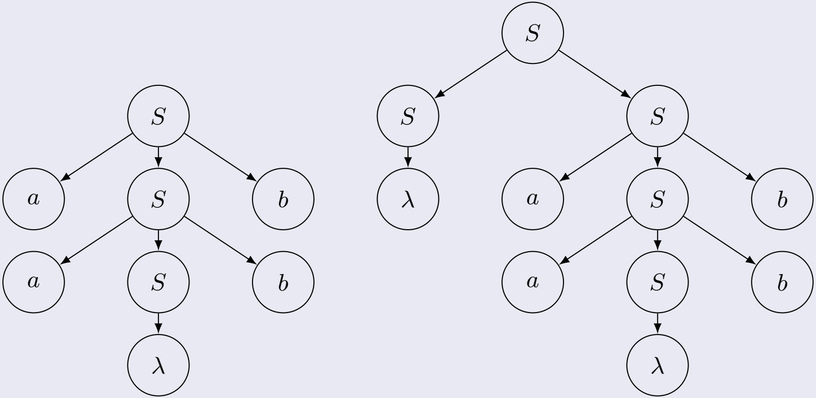
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

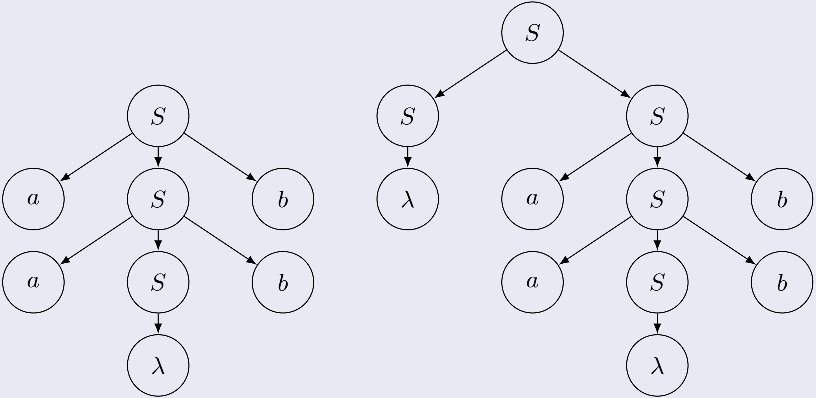
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

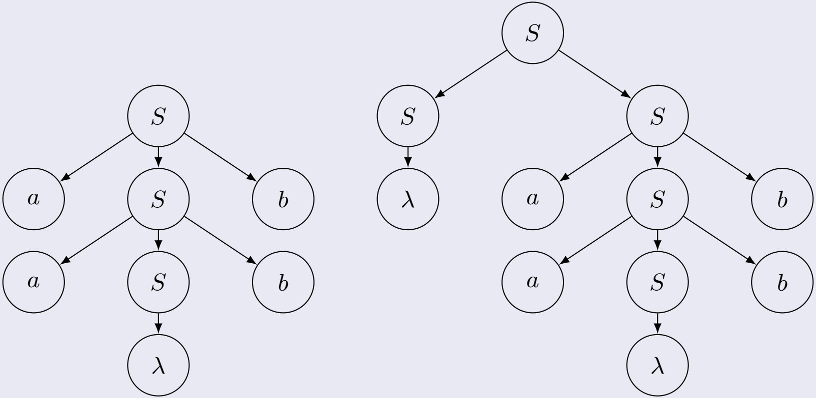
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

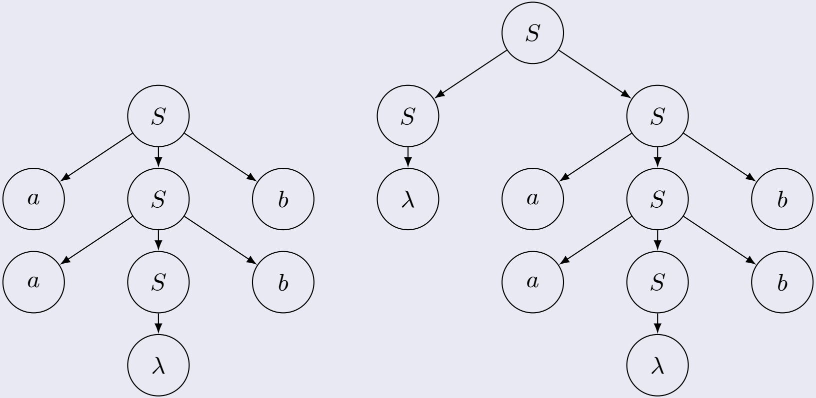
The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.10

The grammar in Example 5.4, with productions $S \to aSb|SS|\lambda$, is ambiguous. The sentence $aabb$ has the two derivation trees shown in the Figure.



Ambiguity is a common feature of natural languages, where it is tolerated and dealt with in a variety of ways. In programming languages, where there should be only one interpretation of each statement, ambiguity must be removed when possible. Often we can achieve this by rewriting the grammar in an equivalent, unambiguous form.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

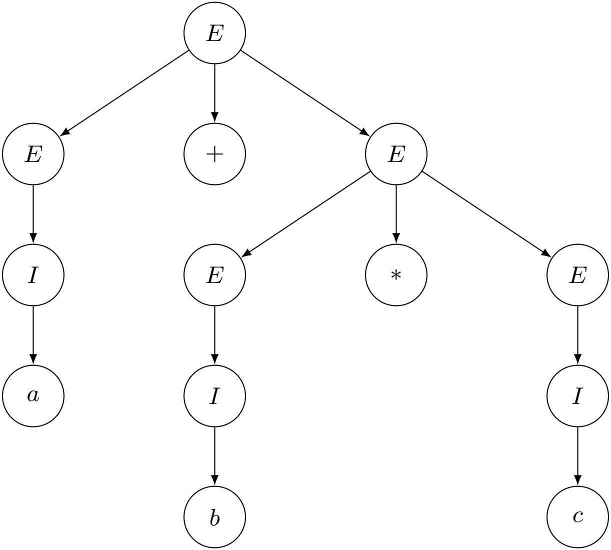
$$V = \{E, I\},$$
$$T = \{a, b, c, +, *, (,)\},$$

and productions

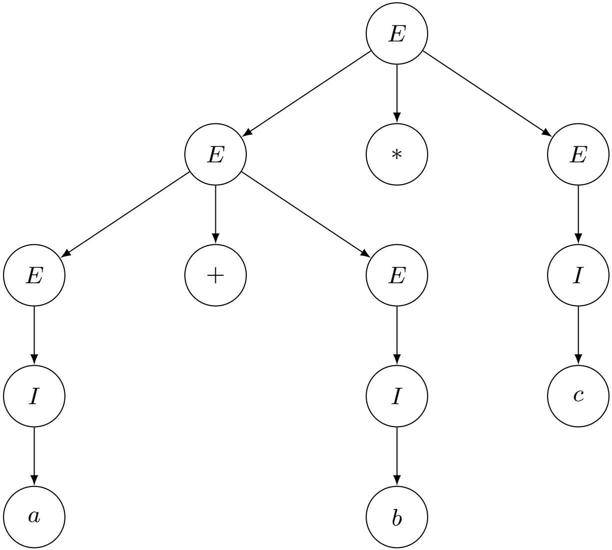
$$E \to I,$$
$$E \to E + E,$$
$$E \to E * E,$$
$$E \to (E),$$
$$I \to a|b|c.$$

The strings $(a + b) * c$ and $a * b + c$ are in $L(G)$. It is easy to see that this grammar generates a restricted subset of arithmetic expressions for C-like programming languages. The grammar is ambiguous. For instance, the string $a + b * c$ has two different derivation trees, as shown in the following two Figures.

E
E
+
E
I
E
*
E
a
I
I
b
c


E
E
$+$
E
I
E
$*$
E
a
I
I
b
c

E
E
*
E
E
+
E
I
I
I
c
a
b


E
E
*
E
E
+
E
I
I
I
c
a
b

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

5.2 Parsing and Ambiguity

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$
\begin{aligned}
E &\to T,\\
T &\to F,\\
F &\to I,\\
E &\to E + T,\\
T &\to T * F,\\
F &\to (E),\\
I &\to a|b|c.
\end{aligned}
$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

5.2 Parsing and Ambiguity

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$
\begin{aligned}
E &\to T, \\
T &\to F, \\
F &\to I, \\
E &\to E + T, \\
T &\to T * F, \\
F &\to (E), \\
I &\to a|b|c.
\end{aligned}
$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

One way to resolve the ambiguity is, as is done in programming manuals, to associate precedence rules with the operators $+$ and $*$. Since $*$ normally has higher precedence than $+$, we would take the first Figure as the correct parsing as it indicates that $b * c$ is a subexpression to be evaluated before performing the addition. However, this resolution is completely outside the grammar. It is better to rewrite the grammar so that only one parsing is possible.

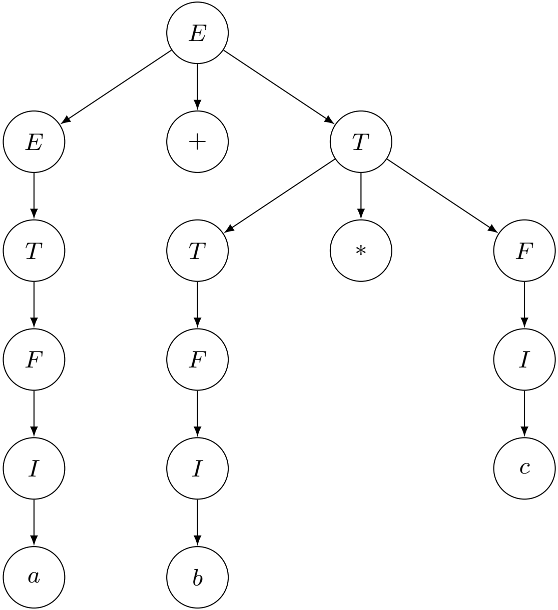
Example 5.12

To rewrite the grammar in Example 5.11 we introduce new variables, taking V as $\{E, T, F, I\}$, and replacing the productions with

$$E \to T,$$
$$T \to F,$$
$$F \to I,$$
$$E \to E + T,$$
$$T \to T * F,$$
$$F \to (E),$$
$$I \to a|b|c.$$

A derivation tree of the sentence $a + b * c$ is shown in the following Figure.

E
E
+
T
T
T
*
F
F
F
I
I
I
c
a
b


E
E
+
T
T
T
*
F
F
F
I
I
I
a
b
c

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.12 (continuation)

No other derivation tree is possible for this string: The grammar is unambiguous. It is also equivalent to the grammar in Example 5.11. It is not too hard to justify these claims in this specific instance, but, in general, the questions of whether a given context-free grammar is ambiguous or whether two given context-free grammars are equivalent are very difficult to answer. In fact, we shall later show that there are no general algorithms by which these questions can always be resolved.

In the foregoing example the ambiguity came from the grammar in the sense that it could be removed by finding an equivalent unambiguous grammar. In some instances, however, this is not possible because the ambiguity is in the language.

Definition 5.6

If L is a context-free language for which there exists an unambiguous grammar, then L is said to be *unambiguous*. If every grammar that generates L is ambiguous, then the language is called *inherently ambiguous*.

It is a somewhat difficult matter even to exhibit an inherently ambiguous language. The best we can do here is give an example with some reasonably plausible claim that it is inherently ambiguous.

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language
$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$
is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that
$$L = L_1 \cup L_2,$$
where L_1 is generated by
$$S_1 \to S_1 c | A,$$
$$A \to aAb | \lambda$$
and L_2 is given by an analogous grammar with start symbol S_2 and productions
$$S_2 \to aS_2 | B,$$
$$B \to bBc | \lambda.$$
Then L is generated by the combination of these two grammars with the additional production
$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language
$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$
is an inherently ambiguous context-free language.
That L is context-free is easy to show. Notice that
$$L = L_1 \cup L_2,$$
where L_1 is generated by
$$S_1 \to S_1 c | A,$$
$$A \to aAb | \lambda$$
and L_2 is given by an analogous grammar with start symbol S_2 and productions
$$S_2 \to aS_2 | B,$$
$$B \to bBc | \lambda.$$
Then L is generated by the combination of these two grammars with the additional production
$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13

The language
$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$
is an inherently ambiguous context-free language.
That L is context-free is easy to show. Notice that
$$L = L_1 \cup L_2,$$
where L_1 is generated by
$$S_1 \to S_1 c | A,$$
$$A \to aAb | \lambda$$
and L_2 is given by an analogous grammar with start symbol S_2 and productions
$$S_2 \to aS_2 | B,$$
$$B \to bBc | \lambda.$$
Then L is generated by the combination of these two grammars with the additional production
$$S \to S_1 | S_2.$$

Example 5.13

The language
$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$
is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that
$$L = L_1 \cup L_2,$$
where L_1 is generated by
$$S_1 \to S_1 c | A,$$
$$A \to aAb | \lambda$$
and L_2 is given by an analogous grammar with start symbol S_2 and productions
$$S_2 \to aS_2 | B,$$
$$B \to bBc | \lambda.$$
Then L is generated by the combination of these two grammars with the additional production
$$S \to S_1 | S_2.$$

Example 5.13

The language

$$L = \{a^n b^n c^m : n, m \geqslant 0\} \cup \{a^n b^m c^m : n, m \geqslant 0\},$$

is an inherently ambiguous context-free language.

That L is context-free is easy to show. Notice that

$$L = L_1 \cup L_2,$$

where L_1 is generated by

$$S_1 \to S_1 c | A,$$

$$A \to aAb | \lambda$$

and L_2 is given by an analogous grammar with start symbol S_2 and productions

$$S_2 \to aS_2 | B,$$

$$B \to bBc | \lambda.$$

Then L is generated by the combination of these two grammars with the additional production

$$S \to S_1 | S_2.$$

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Example 5.13 (continuation)

The grammar is ambiguous since the string $a^n b^n c^n$ has two distinct derivations, one starting with $S \Rightarrow S_1$, the other with $S \Rightarrow S_2$. It does not, of course, follow from this that L is inherently ambiguous as there might exist some other unambiguous grammars for it. But in some way L_1 and L_2 have conflicting requirements, the first putting a restriction on the number of a's and b's, while the second does the same for b's and c's. A few tries will quickly convince you of the impossibility of combining these requirements in a single set of rules that cover the case $n = m$ uniquely. A rigorous argument, though, is quite technical. One proof can be found in Harrison (1978).

Thank You for attention!