CSci 311, Models of Computation
Chapter 5
Context-Free Languages

H. Conrad Cunningham

29 December 2015

Contents

Introduction . . . 1

5.1 Context-Free Grammars . . . 2

5.1.1 Definition of Context-Free Grammars . . . 2

5.1.2 Linz Example 5.1 . . . 2

5.1.3 Linz Example 5.2 . . . 3

5.1.4 Linz Example 5.3 . . . 4

5.1.5 Linz Example 5.4 . . . 4

5.1.6 Leftmost and Rightmost Derivations . . . 5

5.1.7 Linz Example 5.5 . . . 6

5.1.8 Derivation Trees . . . 6

5.1.9 Linz Example 5.6 . . . 7

5.1.10 Relation Between Sentential Forms and Derivation Trees . 7

5.2 Parsing and Ambiguity . . . 9

5.2.1 Generation versus Parsing . . . 9

5.2.2 Exhaustive Search Parsing . . . 9

5.2.3 Linz Example 5.7 . . . 10

5.2.4 Flaws in Exhaustive Search Parsing . . . 11

5.2.5 Linz Example 5.8 . . . 11

5.2.6 Toward Better Parsing Algorithms 12
5.2.7 Simple Grammar Definition 13
5.2.8 Linz Example 5.9 13
5.2.9 Parsing Simple Grammars 13
5.2.10 Ambiguity in Grammars and Languages 14
5.2.11 Linz Example 5.10 14
5.2.12 Linz Example 5.11 14
5.2.13 Linz Example 5.12 16
5.2.14 Inherently Ambiguous 17
5.2.15 Linz Example 5.13 17
5.3 Context-Free Grammars and Programming Languages 18

Acknowledgements: MS student Clay McLeod assisted in preparation of these notes. These lecture notes are for use with Chapter 5 of the textbook: Peter Linz. *Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett Learning, 2012. The terminology and notation used in these notes are similar to those used in the Linz textbook. This document uses several figures from the Linz textbook.

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of December 2015 seems to be a recent version of Firefox from Mozilla.

Introduction

In Linz Section 4.3, we saw that not all languages are regular. We examined the Pumping Lemma for Regular Languages as a means to prove that a specific language is not regular.

In Linz Example 4.6, we proved that

$$L = \{a^n b^n : n \geq 0\}$$

is not regular.

If we let $a =$ "(" and $b =$ ")", then L becomes a language of nested parenthesis.

This language is in a larger family of languages called the *context-free languages*.

Context-free languages are very important because many practical aspects of programming languages are in this family.

In this chapter, we explore the context-free languages, beginning with *context-free grammars*.

One key process we explore is *parsing*, which is the process of determining the grammatical structure of a sentence generated from a grammar.

5.1 Context-Free Grammars

5.1.1 Definition of Context-Free Grammars

Remember the restrictions we placed on *regular grammars* in Linz Section 3.3:

- The *left side* consists of a single variable.
- The *right side* has a special form.

To create more powerful grammars (i.e., that describe larger families of languages), we relax these restrictions.

For context-free grammars, we maintain the left-side restriction but relax the restriction on the right side.

Linz Definition 5.1 (Context-Free Grammar): A grammar $G = (V, T, S, P)$ is *context-free* if all productions in P have the form

$$A \rightarrow x$$

where $A \in V$ and $x \in (V \cup T)^*$. A language L is context-free if and only if there is a context-free grammar G such that $L = L(G)$.

The family of regular languages is a subset of the family of context-free languages!

Thus, *context-free grammars*

- enable the right side of a production to be substituted for a variable on the left side at any time in a sentential form
- with no dependencies on other symbols in the sentential form.

5.1.2 Linz Example 5.1

Consider the grammar $G = (\{S\}, \{a, b\}, S, P)$ with productions:

$$S \rightarrow aSa$$
$$S \rightarrow bSb$$
$$S \rightarrow \lambda$$

Note that this grammar satisfies the definition of context-free.

A possible derivation using this grammar is as follows:

$$S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$$

From this derivation, we see that

$$L(G) = \{ww^R : w \in \{a, b\}^*\}.$$

The language is context-free, but, as we demonstrated in Linz Example 4.8, it is not regular.

This grammar is *linear* because it has at most one variable on the right.

5.1.3 Linz Example 5.2

Consider the grammar G with productions:

$$\begin{aligned} &S \rightarrow abB \\ &A \rightarrow aaBb \\ &B \rightarrow bbAa \\ &A \rightarrow \lambda \end{aligned}$$

Note that this grammar also satisfies the definition of context free.

A possible derivation using this grammar is:

$$\begin{aligned} S &\Rightarrow abB \Rightarrow abbbAa \Rightarrow abbbaaBba \Rightarrow abbbaabbAaba \\ &\Rightarrow abbbaabbaaBbaba \Rightarrow abbbaabbaabbAababa \Rightarrow abbbaabbaabbababa \end{aligned}$$

We can see that:

$$L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}$$

This grammar is also *linear* (as defined in Linz Section 3.3). Although linear grammars are context free, not all context free grammars are linear.

5.1.4 Linz Example 5.3

Consider the language

$$L = \{a^n b^m : n \neq m\}.$$

This language is context free. To show that this is the case, we must construct a context-free grammar that generates the language

First, consider the $n = m$ case. This can be generated by the productions:

$S \rightarrow aSb \mid \lambda$

Now, consider the $n > m$ case. We can modify the above to generate extra a's on the left.

$S \rightarrow AS_1$
$S_1 \rightarrow aS_1b \mid \lambda$
$A \rightarrow aA \mid a$

Finally, consider the $n < m$ case. We can further modify the grammar to generate extra b's on right.

$S \rightarrow AS_1 \mid S_1B$
$S_1 \rightarrow aS_1b \mid \lambda$
$A \rightarrow aA \mid a$
$B \rightarrow bB \mid b$

This grammar is context free, but it is *not linear* because the productions with S on the left are not in the required form.

Although this grammar is not linear, there exist other grammars for this language that are linear.

5.1.5 Linz Example 5.4

Consider the grammar with productions:

$S \rightarrow aSb \mid SS \mid \lambda$

This grammar is also context-free but not linear.

Example strings in $L(G)$ include $abaabb$, $aababb$, and $ababab$. Note that:

- a and b are always generated in pairs.
- a precedes the matching b.
- A prefix of a string may contain several more a's than b's.

We can see that $L(G)$ is

$\{\ w \in \{a, b\}^* : n_a(w) = n_b(w)$ and $n_a(v) \geq n_b(v)$ for any prefix v of $w\ \}$.

What is a programming language connection for this language?

- Let $a =$ "(" and $b =$ ")".
- This gives us a language of properly nested parentheses.

5.1.6 Leftmost and Rightmost Derivations

Consider grammar $G = (\{A, B, S\}, \{a, b\}, S, P)$ with productions:

$S \to AB$
$A \to aaA$
$A \to \lambda$
$B \to Bb$
$B \to \lambda$

This grammar generates the language $L(G) = \{a^{2n}b^m : n \geq 0, m \geq 0\}$.

Now consider the two derivations:

$$S \Rightarrow AB \Rightarrow aaAB \Rightarrow aaB \Rightarrow aaBb \Rightarrow aab$$

$$S \Rightarrow AB \Rightarrow ABb \Rightarrow aaABb \Rightarrow aaAb \Rightarrow aab$$

These derivations yield the same sentence using exactly the same productions. However, the productions are applied in different orders.

To eliminate such incidental factors, we often require that the variables be replaced in a specific order.

Linz Definition 5.2 (Leftmost and Rightmost Derivations): A derivation is *leftmost* if, in each step, the leftmost variable in the sentential form is replaced. If, in each step, the rightmost variable is replaced, then the derivation is *rightmost*.

5.1.7 Linz Example 5.5

Consider the grammar with productions:

$S \to aAB$
$A \to bBb$
$B \to A \mid \lambda$

A leftmost derivation of the string *abbbb* is:

$$S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$$

Similarly, a rightmost derivation of the string *abbbb* is:

$$S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb$$

5.1.8 Derivation Trees

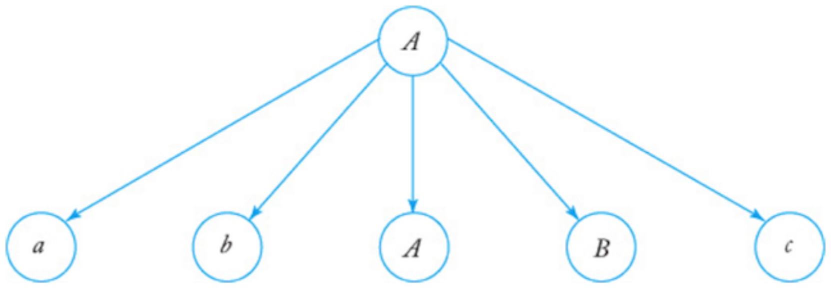
We can also use a *derivation tree* to show derivations in a manner that is independent of the order in which productions are applied.

A derivation tree is an ordered tree in which we label the nodes with the left sides of productions and the children of a node represent its corresponding right sides.

The production

$$A \rightarrow abABc$$

is shown as a derivation tree in Linz Figure 5.1.



Linz Fig. 5.1: Derivation Tree for Production $A \rightarrow abABc$

Linz Definition 5.3 (Derivation Tree): Let $G = (V, T, S, P)$ be a context-free grammar. An ordered tree is a *derivation tree* for G if and only if it has the following properties:

1. The root is labeled S.
2. Every leaf has a label from $T \cup \{\lambda\}$.
3. Every interior vertex (i.e., a vertex that is not a leaf) has a label from V.

4. If a vertex has a label $A \in V$, and its children are labeled (from left to right) $a_1, a_2, \cdots, a_n$, then P must contain a production of the form $A \rightarrow a_1 a_2 \cdots\ a_n$.
5. A leaf labeled λ has no siblings, that is, a vertex with a child labeled λ can have no other children.

If properties 3, 4, and 5 and modified property 2 (below) hold for a tree, then it is a *partial derivation tree*.

2. (modified) Every leaf has a label from $V \cup T \cup \{\lambda\}$

If we read the leaves of a tree from left to right, omitting any λ's encountered, we obtain a string called the *yield* of the tree.

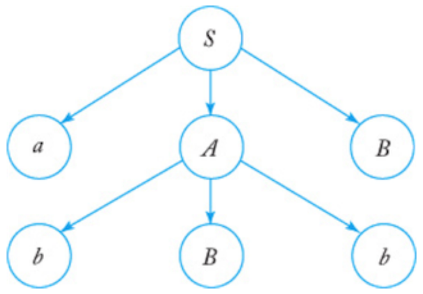
The descriptive term from *left to right* means that we traverse the tree in a depth-first manner, always exploring the leftmost unexplored branch first. The *yield* is the string of terminals encountered in this traversal.

5.1.9 Linz Example 5.6

Consider the grammar G with productions:

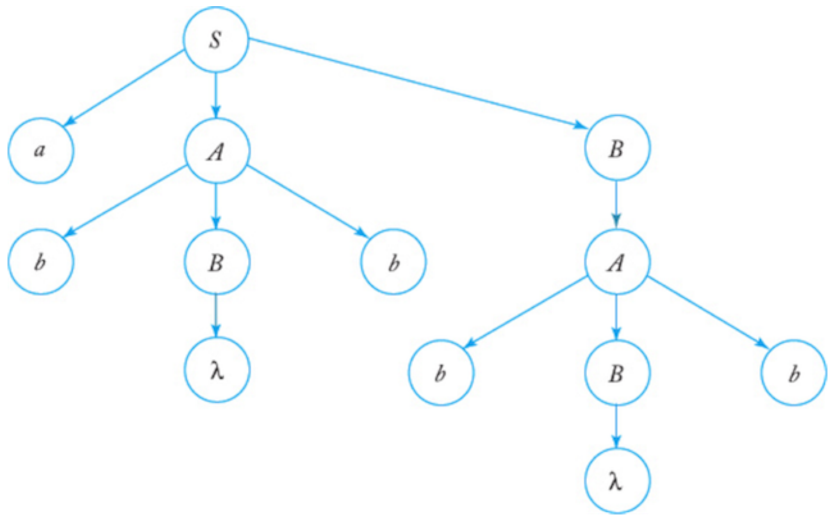
$S \rightarrow aAB$
$A \rightarrow bBb$
$B \rightarrow A \mid \lambda$

Linz Figure 5.2 shows a partial derivation tree for G with the yield $abBbB$. This is a *sentential form* of the grammar G.



Linz Fig. 5.2: Partial Derivation Tree

Linz Figure 5.3 shows a derivation tree for G with the yield $abbbb$. This is a *sentence* of $L(G)$.



Linz Fig. 5.3: Derivation Tree

5.1.10 Relation Between Sentential Forms and Derivation Trees

Derivation trees give explicit (and visualizable) descriptions of derivations. They enable us to reason about context-free languages much as transition graphs enable use to reason about regular languages.

Linz Theorem 5.1 (Connection between Derivations and Derivation Trees): Let $G = (V, T, S, P)$ be a context-free grammar. Then the following properties hold:

- For every $w \in L(G)$, there exists a derivation tree of G whose yield is w.
- The yield of any derivation tree of G is in $L(G)$.
- If t_G is any partial derivation tree for G whose root is labeled S, then the yield of t_G is a sentential form of G.

Proof: See the proof in the Linz textbook.

Derivation trees:

- show which productions are used to generate a sentence
- abstract out the order in which individual productions are applied
- enable the construction of eiher a leftmost or rightmost derivation

5.2 Parsing and Ambiguity

5.2.1 Generation versus Parsing

The previous section concerns the generative aspects of grammars–using a grammar to generate strings in the language.

This section concerns the analytical aspects of grammars–processing strings from the language to determine their derivations. This process is called *parsing*.

For example, a compiler for a programming language must parse a program (i.e., a sentence in the language) to determine the derivation tree for the program.

- This verifies that the program is indeed in the language (syntactically).
- Construction of the derivation tree is needed to execute the program (e.g., to generate the machine-level code corresponding to the program).

5.2.2 Exhaustive Search Parsing

Given some $w \in L(G)$, we can parse w with respect to grammar G by:

- systematically constructing all derivations
- determining whether any derivation matches w

This is called *exhaustive search parsing* or *brute force parsing*. A more complete description of the algorithm is below.

This is a form of *top-down parsing* in which a derivation tree is constructed from the root downward.

Note: An alternative approach is *bottom-up parsing* in which the derivation tree is constructed from leaves upward. Bottom-up parsing techniques often have limitations in terms of the grammars supported but often give more efficient algorithms.

Exhaustive Search Parsing Algorithm

```
– Add root and 1st level of all derivation trees
$F \leftarrow \{x : s \rightarrow x$ in $P$ of $G\}$
while $F \neq \emptyset$ and $w \notin F$ do
    $F' \leftarrow \emptyset$
    – Add next level of all derivation trees
    for all $x \in F$ do
        if $x$ can generate $w$ then
            $V \leftarrow$ leftmost variable of $x$
            for all productions $V \rightarrow y$ in $G$ do
            $F' \leftarrow F' \cup \{x'\}$ where $x' = x$ with $V \leftarrow y$
    $F \leftarrow F'$
```

Note: The above algorithm determines whether a string w is in $L(G)$. It can be modified to build the actual derivation or derivation tree.

5.2.3 Linz Example 5.7

Note: The presentation here uses the algorithm above, rather than the approach in the Linz textbook.

Consider the grammar G with productions:

$$S \rightarrow SS \mid aSb \mid bSa \mid \lambda$$

Parse the string $w = aabb$.

After initialization: $F = \{SS, aSb, bSa, \lambda\}$ (from the righthand sides of the grammar's four productions with S on the left).

First iteration: The loop test is true because F is nonempty and w is not present.

The algorithm does not need to consider the sentential forms bSa and λ in F because neither can generate w.

The inner loop thus adds $\{SSS, aSbS, bSaS, S\}$ from the leftmost derivations from sentential form SS and also adds $\{aSSb, aaSbb, abSab, ab\}$ from the leftmost derivations from sentential form aSb.

Thus $F = \{SSS, aSbS, bSaS, S, aSSb, aaSbb, abSab, ab\}$ at the end of the first iteration.

Second iteration: The algorithm enters the loop a second time because F is nonempty and does not contain w.

The algorithm does not need to consider any sentential form beginning with b or ab, thus eliminating $\{bSaS, abSab, ab\}$ and leaving $\{SSS, aSbS, S, aSSb, aaSbb\}$ of interest.

This iteration generates 20 new sentential forms from applying each of the 4 productions to each of the 5 remaining sentential forms.

In particular, note that that sentential form $aaSbb$ yields the target string $aabb$ when production $S \rightarrow \lambda$ is applied.

Third iteration: The loop terminates because w is present in F.

Thus we can conclude $w \in L(G)$.

5.2.4 Flaws in Exhaustive Search Parsing

Exhaustive search parsing has serious flaws:

- It is tedious and inefficient.
- It might not terminate when $w \notin L(G)$.

For example, if we choose $w = abb$ in the previous example, the algorithm goes into an infinite loop.

The fix for nontermination is to ensure sentential forms increase in length for each production. That is, we eliminate productions of forms:

$$A \rightarrow \lambda$$
$$A \rightarrow B$$

Chapter 6 of the Linz textbook (which we will not cover this semester) shows that this does not reduce the power of the grammar.

5.2.5 Linz Example 5.8

Consider the grammar with productions:

$$S \rightarrow SS \mid aSb \mid bSA \mid ab \mid ba$$

This grammar generates the same language as the one in Linz Example 5.7 above, but it satisfies the restrictions given in the previous subsection.

Given any nonempty string w, exhaustive search will terminate in no more than $|w|$ rounds for such grammars.

5.2.6 Toward Better Parsing Algorithms

Linz Theorem 5.2 (Exhaustive Search Parsing): Suppose that $G = (V, T, S, P)$ is a context-free grammar that does not have any rules of one of the forms

$$A \rightarrow \lambda$$
$$A \rightarrow B$$

where $A, B \in V$. Then the exhaustive search parsing method can be formulated as an algorithm which, for any $w \in T*$, either parses w or tells us that parsing is impossible.

Proof outline

- Each production must increase either the length or number of terminals.
- The maximum length of a sentential form is $|w|$, which is the maximum number of terminal symbols.
- Thus for some w, the number of loop iterations is at most $2|w|$.

But exhaustive search is *still inefficient.* The number of sentential forms to be generated is

$$\sum_{i=1}^{2|w|} |P|^i.$$

That is, it grows exponentially with the length of the string.

Linz Theorem 5.3 (Efficient Parsing): For every context-free grammar there exists an algorithm that parses any $w \in L(G)$ in a number of steps proportional to $|w|^3$.

- Construction of more efficient context-free parsing methods is left to compiler courses.
- $|w|^3$ is still inefficient.
- We would prefer linear ($|w|$) parsing.
- Again we must restrict the grammar in our search for more efficient parsing. The next subsection illustrates on such grammar.

5.2.7 Simple Grammar Definition

Linz Definition 5.4 (Simple Grammar): 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.

5.2.8 Linz Example 5.9

The grammar

$$S \to aS \mid bSS \mid c$$

is an s-grammar.

The grammar

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

is *not* an s-grammar because (S, a) occurs twice.

5.2.9 Parsing Simple Grammars

Although s-grammars are quite restrictive, many features of programming languages can be described with s-grammars (e.g., grammars for arithmetic expressions).

If G is s-grammar, then $w \in L(G)$ can be parsed in linear time.

To see this, consider string $w = a_1 a_2 \cdots a_n$ and use the exhaustive search parsing algorithm.

1. The s-grammar has at most one rule with a_1 on left: $S \to a_1 A_1 A_2 \cdots$. *Choose it!*
2. Then the s-grammar has at most one rule with a_2 on left: $A_1 \to a_2 B_1 B_2 \cdots$. *Choose it!*
3. And so forth up to the nth terminal.

The number of steps is proportional to $|w|$ because each step consumes one symbol of w.

5.2.10 Ambiguity in Grammars and Languages

A derivation tree for some string generated by a context-free grammar may not be unique.

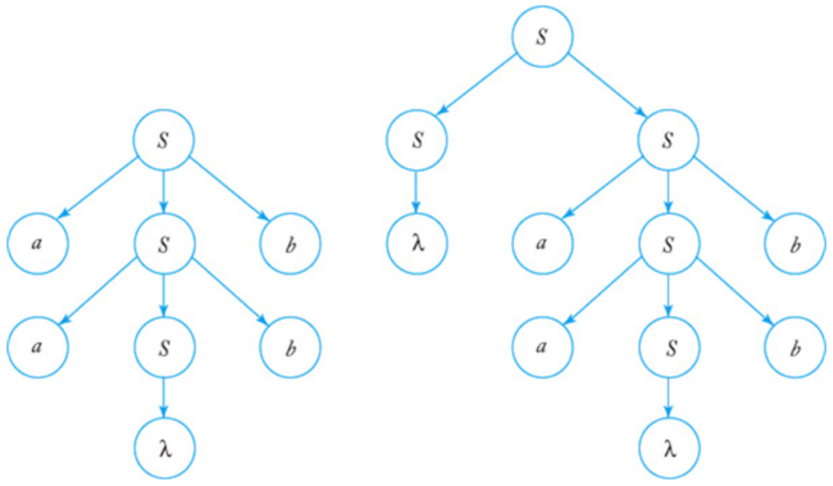
Linz Definition 5.5 (Ambiguity): A context-free grammar G is said to be *ambiguous* if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively, ambiguity implies the existence of two or more leftmost or rightmost derivations.

5.2.11 Linz Example 5.10

Again consider the grammar in Linz Example 5.4. Its productions are

$$S \rightarrow aSb \mid SS \mid \lambda.$$

The string $w = aabb$ has two derivation trees as shown in Linz Figure 5.4



Linz Fig. 5.4: Two Derivation Trees for $aabb$

The left tree corresponds to the leftmost derivation $S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

The right tree corresponds to the leftmost derivation $S \Rightarrow SS \Rightarrow \lambda S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$.

Thus the grammar is ambiguous.

5.2.12 Linz Example 5.11

Consider the grammar $G = (V, T, E, P)$ with

$$V = \{E, I\}$$
$$T = \{a, b, c, +, *, (,)\}$$

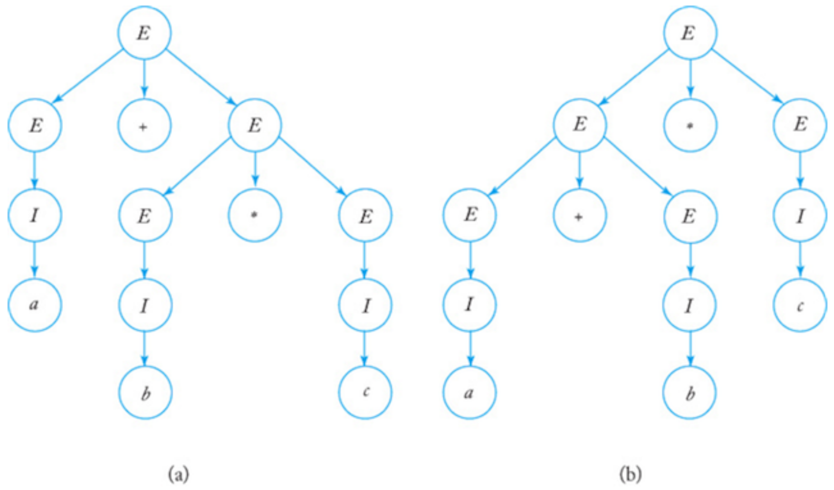
and P including the productions:

$$E \rightarrow I$$
$$E \rightarrow E + E$$
$$E \rightarrow E * E$$

$$E \to (E)$$
$$I \to a \mid b \mid c$$

This grammar generates a subset of the arithmetic expressions for a language like C or Java. It contains strings such as $(a+b)*c$ and $a*b+c$.

Linz Figure 5.5 shows two derivation trees for the string $a+b*c$. Thus this grammar is ambiguous.



Linz Fig. 5.5: Two Derivation Trees for $a+b*c$

Why is ambiguity a problem?

Remember that the semantics (meaning) of the expression is also associated with the structure of the expression. The structure determines how the (machine language) code is generated to carry out the computation.

How do real programming languages resolve this ambiguity?

Often, they add *precedence rules* that give priority to "$*$" over "$+$". That is, the multiplication operator binds more tightly than addition.

This solution is totally outside the world of the context-free grammar. It is, in some sense, a hack.

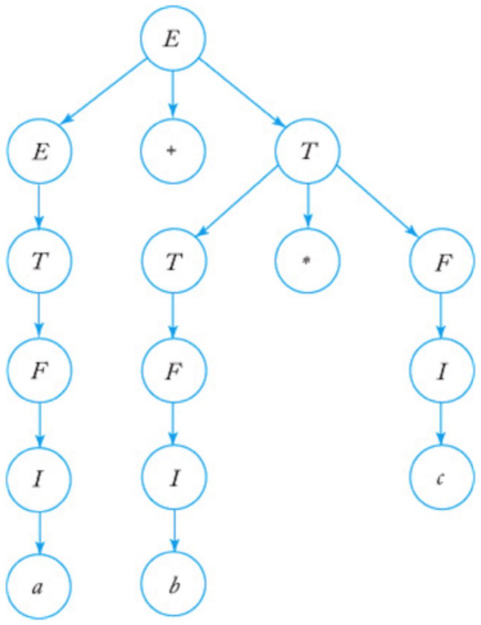
A better solution is to rewrite the grammar (or sometimes redesign te language) to eliminate the ambiguity.

5.2.13 Linz Example 5.12

To rewrite the grammar in Linz Example 5.11, we introduce new variables, making V the set $\{E, T, F, I\}$, and replacing the productions with the following:

$E \to T$
$T \to F$
$F \to I$
$E \to E + T$
$T \to T * F$
$F \to (E)$
$I \to a \mid b \mid c$

Linz Figure 5.6 shows the only derivation tree for string $a + b * c$ in this revised grammar for arithmetic expressions.



Linz Fig. 5.6: Derivation Tree for $a + b * c$ in Revised Grammar

5.2.14 Inherently Ambiguous

Linz 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 language is called *inherently ambiguous*.

It is difficult to demonstrate that a grammar is inherently ambiguous. Often the best we can do is to give examples and argue informally that all grammars must be ambiguous.

5.2.15 Linz Example 5.13

The language

$$L = \{a^n b^n c^m\} \cup \{a^n b^m c^m\},$$

with n and m non-negative, is an inherently ambiguous context-free language.

Note that $L = L_1 \cup L_2$.

We can generate L_1 with the context-free grammar:

$$\begin{array}{l} S_1 = S_1 c \mid A \\ A \to aAb \mid \lambda \end{array}$$

Similarly, we can generate L_2 with the context-free grammar:

$$\begin{array}{l} S_2 = aS_2 \mid B \\ B \to bBc \mid \lambda \end{array}$$

We can thus construct the union of these two sublanguages by adding a new production:

$$S \to S_1 \mid S_2$$

Thus this is a context-free language.

But consider a string of the form $a^n b^n c^n$ (i.e., $n = m$). It has two derivations, one starting with

$$S \Rightarrow S_1$$

and another starting with

$$S \Rightarrow S_2.$$

Thus the grammar is ambiguous.

L_1 and L_2 have conflicting requirements. L_1 places restrictions on the number of a's and b's while L_2 places restrictions on the number of b's and c's. It is imposible to find production rules that satisfy the $n = m$ case uniquely.

5.3 Context-Free Grammars and Programming Languages

The syntax for practical programming language syntax is usually expressed with context-free grammars. Compilers and interpreters must parse programs in these language to execute them.

The grammar for programming languages is often expressed using the *Backus-Naur Form (BNF)* to express productions.

For example, the language for arithmetic expressing in Linz Example 5.12 can be written in BNF as:

```
    <expression> ::= <term>   | <expression> + <term>
          <term> ::= <factor> | <term> * <factor>
```

The items in angle brackets are variables, the symbols such as "+" and "-" are terminals, the "|" denotes alternatives, and "::=" separates the left and right sides of the productions.

Programming languages often use restricted grammars to get linear parsing: e.g., regular grammars, s-grammars, LL grammars, and LR grammars.

The aspects of programming languages that can be modeled by context-free grammars are called the the *syntax*.

Aspects such as type-checking are not context-free. Such issues are sometimes considered (incorrectly in your instructor's view) as part of the *semantics* of the language.

These are really still syntax, but they must be expressed in ways that are not context free.