CSE396 Lecture Thu. 3/18: Parsing and (Un)Ambiguity

Last lecture ended with saying that context-free grammars were a boon to the development of programming languages. A motivating example starts with the goal of capturing the syntax of numerical
expressions the way we (humans) like to write them.

Example: Numerical Expressions

Let's use BNF notation style for this one, and let's call the start symbol E.

E ::= $(E+E) \mid (E-E) \mid (E*E) \mid (E/E) \mid$ <variable> | <constant>
<variable> ::= *any alphanumeric legal identifier*
<constant> ::= *any legal numeric literal.*

$$E \implies (E+E) \implies ((E-E)+E) \implies^2 ((a-E)+E) \implies^4 ((a-b)+c)\ .$$
$$E \implies (E-E) \implies (E-(E+E)) \implies^6 (a-(b+c))\ .$$
$$E \implies (E-E) \implies^2 (a-E) \implies (a-(E+E)) \implies^4 (a-(b+c))\ .$$

The terminal alphabet Σ includes the parentheses, the operator symbols +,-,*,/, and whatever letters and digits and other punctuation are allowed in variables and constants. We don't want to have to specify the last of these. What we could do is treat the *tokens* <var> and <const> as if they were members of Σ. The text gets around this issue by pretending that a is the only variable and ignoring constants, but being "a little more real" won't hurt us. What we actually do is allow <var> and <const> to derive any legal identifier or constant in one step.

Now the above grammar generates only *fully parenthesized* expressions. It doesn't let you write $a - b + c$ or even $xy + z$. We can get them if we make the parentheses optional:

E ::= $E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid$ <var> | <const>

Now we can derive them---note I write $E \implies^2 a$ to shortcut $E \implies$ <var> $\implies a$, etc.

$$E \implies E-E \implies^2 a-E \implies a-E+E \implies^2 a-b+E \implies^2 a-b+c\ .$$

$$E \implies E*E \implies^2 x*E \implies x*E+E \implies^2 x*y+E \implies^2 x*y+z\ .$$

Does anything about these derivations trouble you? I will say that this "liberal" grammar G generates all and only legal numeric expressions, but it "tells fibs" while doing so. The **sentiential form** $a-E$ seems to say that the whole rest of the expression gets subtracted from a, but that is not how we read the expression $a - b + c$ under the **left-to-right associativity** rule. More clearly (but less insidiously), the sentiential form $x*E$ seems to say that x will multiply both terms in the expression $y+z$ derived from that E, but it only multiplies y in $xy+z$. (Note that you can write $x*(y+z)$ where the

$(y+z)$ part is counted as a *factor*.) Perhaps most insidiously, what about the expression $a/b*c$? You might read it as if the intent were $\frac{a}{bc}$ but it will get **parsed** as $(a/b)*c$ because $/$ and $*$ have equal **precedence**---at least in C/C++/Java/Python/etc. How can we write a grammar to reflect precedence (and associativity)? The answer is to add variables for the extra **syntactic categories** "term" and "factor":

E ::= T | $E+T$ | $E-T$
T ::= F | $T*F$ | T/F
F ::= (E) | <var> | <const>

Now if we try to imitate the first derivation above by putting the minus sign $-$ in first, we get:

$$E \implies E-T \implies T-T \implies F-T \implies^2 a-T$$

and we're stuck: there isn't a rule with $+$ for T. To get $a-b+c$ we now must do

$$E \implies E+T \implies E-T+T \implies T-T+T \implies F-T+T \implies^2 a-T+T \implies^6 a-b+c.$$

Note: You can also do $E \implies T \implies F \implies (E) \implies (E+T)$ and thus get fully-parenthesized expressions too. But you cannot get the sentential form $(E+E)$ from E.

The sentential form $T-T+T$ reads the three terms left-to-right (even though the leftmost term was derived last) at equal level, rather than grouping the last two. Likewise, the only way to derive $xy+z$ is by putting out the $+$ first rather than the $*$ first as before---in terms you may have heard already, the $+$ is the "topmost" or "outermost" operator. The derivation

$$E \implies E+T \implies T+T \implies T*F+T \implies F*F+T \implies^4 x*y+T \implies^3 x*y + z$$

now makes clear that x was never intended to multiply z. We can also still write the fully-parenthesized forms if we wish, as well as options in-between, even silly but legal ones like $(x*(y) + ((z)))$. We can also tack on more syntactic categories, such as having a <factor> involve powers. Some programming languages have a native operation for powers like $**$, but you have to be careful that it is **right-associative**: $a**b**c$ means $a**(b**c) = a^{b^c}$, not $(a**b)**c = (a^b)^c$ because the latter just becomes a^{bc}. In practice, the part of the grammar for expressions in modern programming languages has a dozen or two dozen variables (i.e., syntactic categories). But the point is that not only is the grammar able perfectly to describe the **syntax** of the language (still falling short of checking consistency of types and the number/sequence of arguments in function/method calls, which Ada95 for one called the "semantic" phase), the grammar also is instrumental to write the compiler's **parsing** stage. So let's move on to parsing---still in section 2.1 but not intending to go into the compiler-level detail of the later section 2.4.

Parse Trees, Leftmost Derivations, and Ambiguity.

Definition: A **parse tree** of a CFG $G = (V, \Sigma, \mathcal{R}, S)$ is a finite **rooted tree** in which:

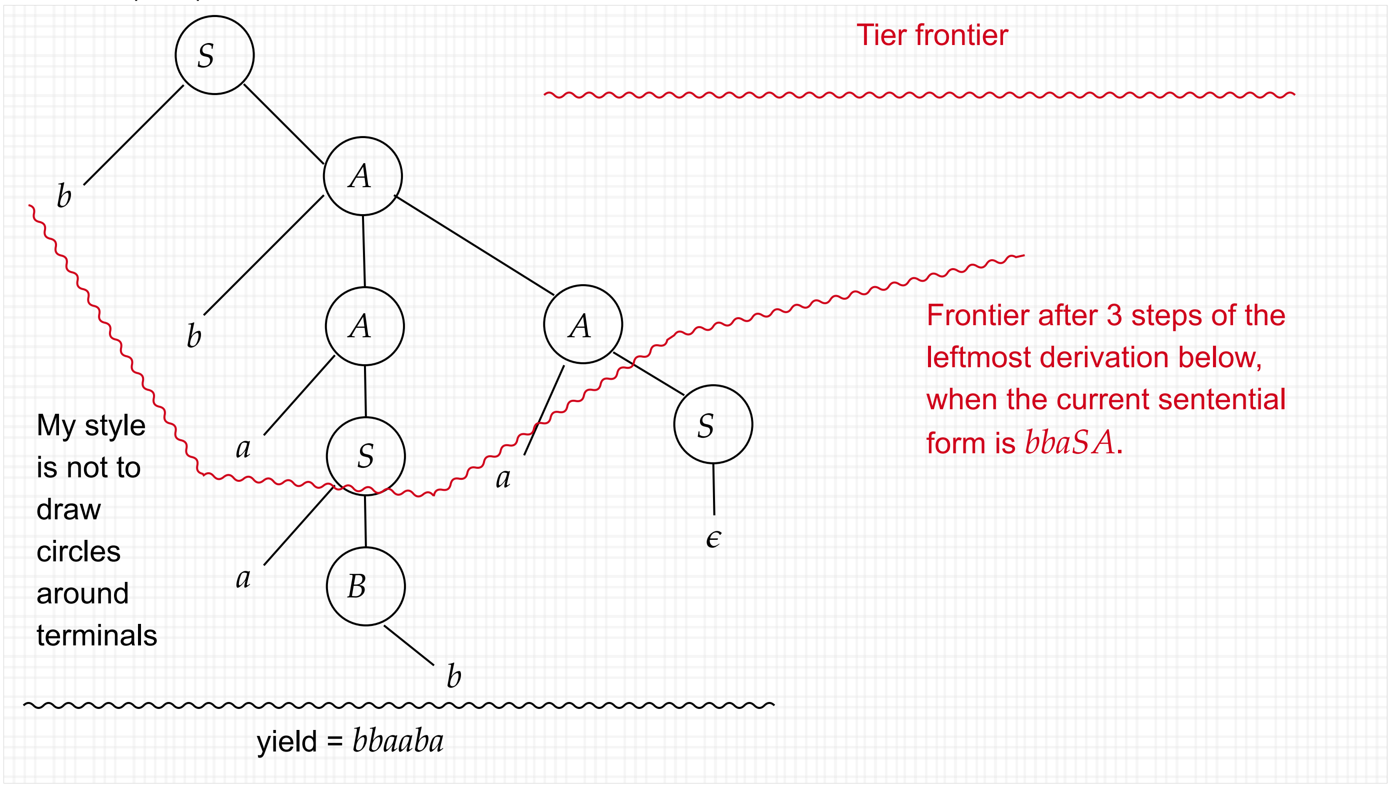
- every leaf is labeled by a terminal symbol $c \in \Sigma$ or by ϵ,
- every internal node is labeled by a variable,
- the root is labeled by S (or by whatever variable we want to derive from), and
- if the children of an internal node with label A are $X_1, X_2, \dots, X_m$ in left-to-right order, where m is the valence of the node, then $A \to X_1 X_2 \cdots X_m$ is a rule in $\mathcal{R}$.

The **yield** of the tree is the string $x \in \Sigma^*$ formed by concatenating the leaves in left-to-right order.

Rooted tree means that one node is distinguished as the root and all other nodes are "below" it (trees grow down not up). The definition of **subtree** is usually restricted to mean taking an internal node A and including *all* nodes below A. My including the clause in (...) means that any subtree T' of a parse tree T can be called a parse tree "rooted at A" by itself. An opposite notion of subtree T'' includes the root and is obtained by deleting zero or more subtrees rooted at internal nodes A except for A itself, so that A effectively becomes a leaf in T''. When the root is S, the yield X of T'' is always a **sentential form**, meaning $S \Longrightarrow_G^* X$. I will refer to the bottoms of such trees as "**tiers**".

Example

$S \;\to\; \epsilon \mid aB \mid bA$
$A \;\to\; a \mid aS \mid bAA$
$B \;\to\; b \mid bS \mid aBB$


Tier frontier
Frontier after 3 steps of the leftmost derivation below, when the current sentential form is $bbaSA$.
My style is not to draw circles around terminals
yield = $bbaaba$

$$S \Longrightarrow bA \Longrightarrow bbAA \Longrightarrow bbaSA \Longrightarrow bbaaBA \Longrightarrow bbaabA \Longrightarrow bbaabaS \Longrightarrow bbaaba.$$

Definition: A derivation is **leftmost** if it always expands the leftmost variable at any step.

We can get a leftmost derivation from a parse tree T by doing a left-to-right transversal of T. (The transversal is considered **preorder** rather than **inorder** or **postorder**, but what matters is its going left-to-right.) From the above tree we get:

$$S \Longrightarrow bA \Longrightarrow bbAA \Longrightarrow bbaSA \Longrightarrow bbaaBA \Longrightarrow bbaabA \Longrightarrow bbaabaS \Longrightarrow bbaaba.$$

[A derivation is **rightmost** if it always expands the rightmost variable (instead). For example:

$$S \Longrightarrow bA \Longrightarrow bbAA \Longrightarrow bbAaS \Longrightarrow bbAa \Longrightarrow bbaSa \Longrightarrow bbaaBa \Longrightarrow bbaaba$$

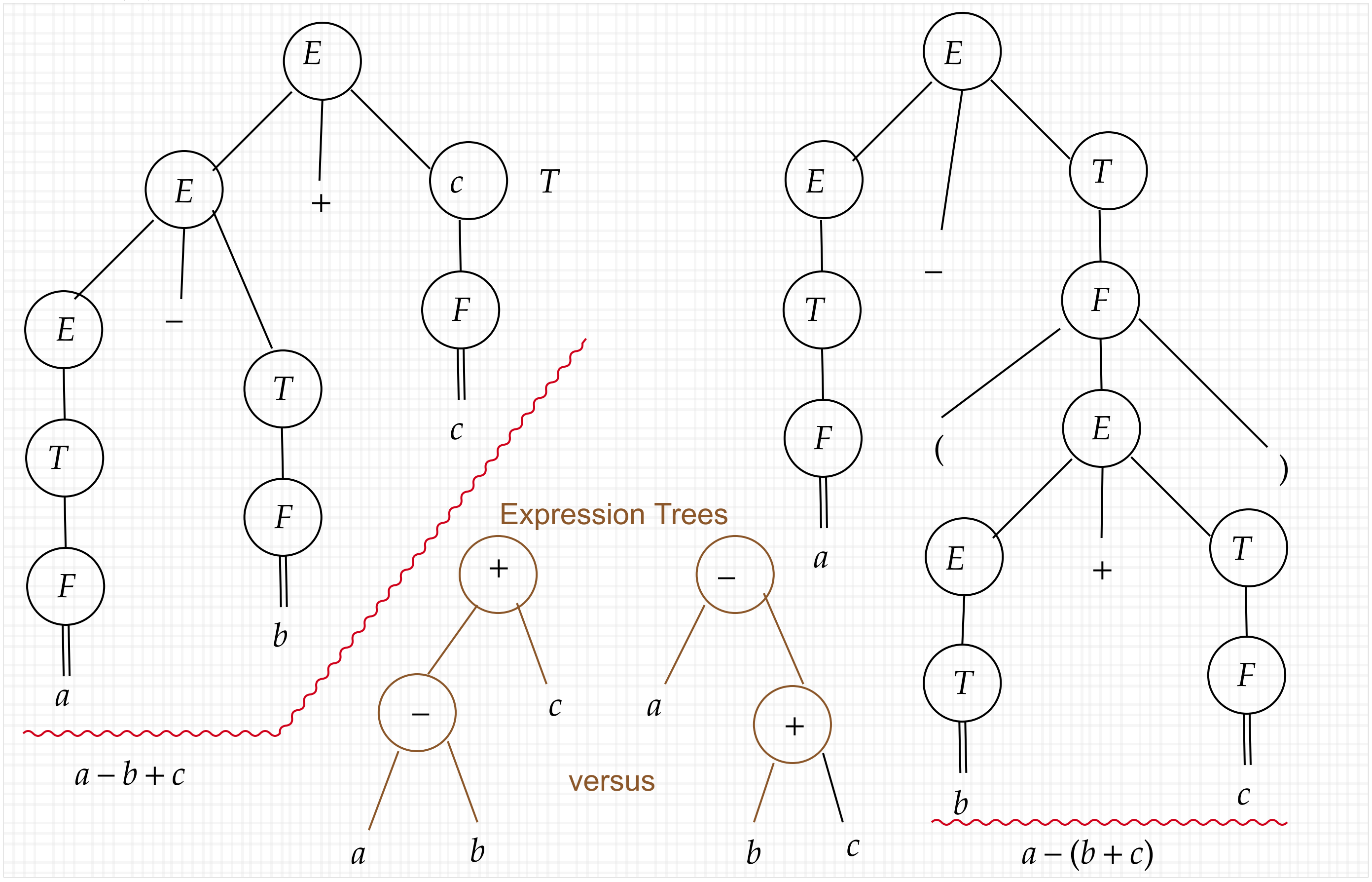
One central family of C-style compilers favored rightmost over leftmost derivations.]

Example: Our expression grammar again:

E ::= T | $E+T$ | $E-T$

T ::= F | $T*F$ | T/F

F ::= (E) | <var> | <const>



$$E \Longrightarrow E+T \Longrightarrow E-T+T \Longrightarrow T-T+T \Longrightarrow F-T+T \Longrightarrow^2 a-T+T \Longrightarrow^6 a-b+c.$$

$$E \Longrightarrow E-T \Longrightarrow T-T \Longrightarrow F-T \Longrightarrow^2 a-T \Longrightarrow a-F \Longrightarrow a-(E) \Longrightarrow a-(E+T)$$

$$\Longrightarrow a-(T+T) \Longrightarrow a-(F+T) \Longrightarrow^2 a-(b+T) \Longrightarrow^2 a-(b+c).$$

Lemma: Parse trees are is 1-to-1 correspondence with **leftmost** derivations. ☒

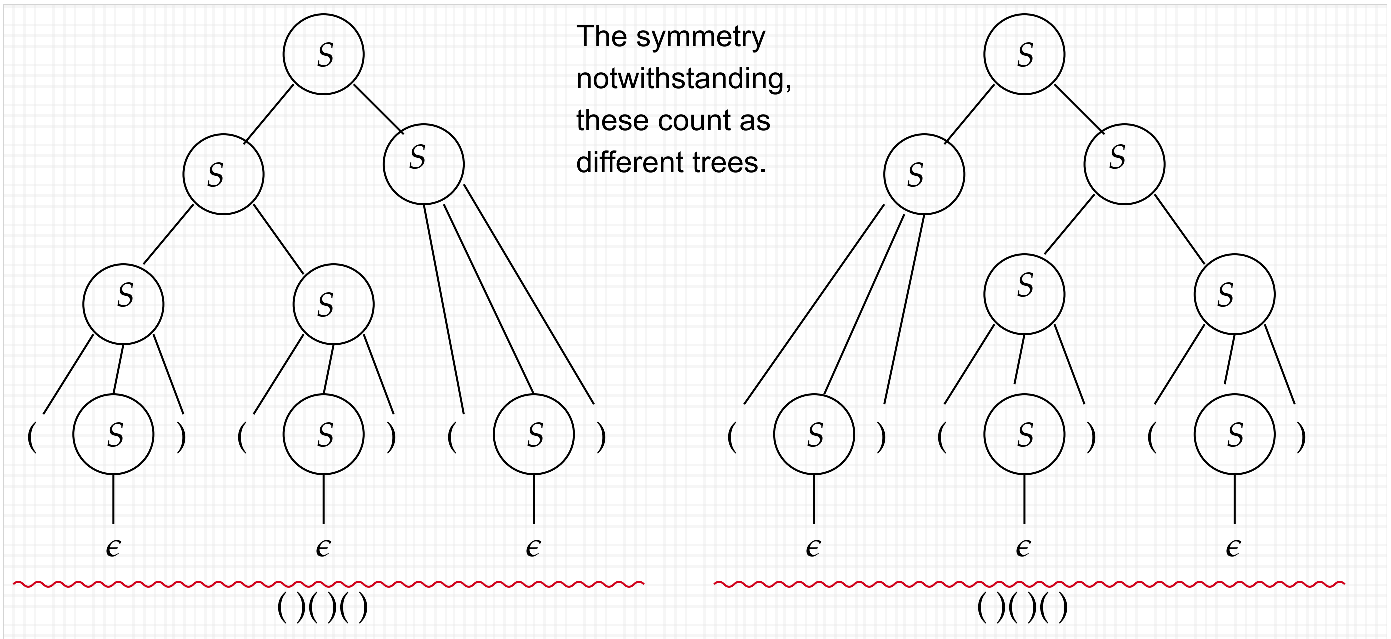
[They are also in 1-to-1 correspondence with **rightmost** derivations.]

Definition: A string $x \in L(G)$ is **ambiguous** in G if it has two different parse trees---equivalently, if x has two different *leftmost* derivations. [And equivalently, if it has two different rightmost derivations.] One ambiguous terminal string makes G itself **ambiguous**. But if G has no ambiguous strings then G is **unambiguous**.

Call a variable A **deadwood** if $L_A = \emptyset$, that is, if A does not derive any terminal string. That means if A appears in a tier T'' then it cannot be completed to a parse tree (hence the name). Otherwise, A is **live**.

Proposition: Any grammar with the rules $A \longrightarrow AA$ or $E \longrightarrow E + E$ for live variables A or E is ambiguous.

We can essentially prove this via the example of the balanced-parentheses grammar $S \longrightarrow SS \mid (S) \mid \epsilon$.



The symmetry notwithstanding, these count as different trees.

$S \Rightarrow SS \Rightarrow SSS \Rightarrow (S)SS \Rightarrow ()SS \Rightarrow^4 ()()()$ $\quad$ $S \Rightarrow SS \Rightarrow (S)S \Rightarrow ()S \Rightarrow ()SS \Rightarrow^4 ()()()$

The case of $E \longrightarrow E + E$ is similar: if y is any string derived from E, then $y + y + y$ has the leftmost derivations $E \Longrightarrow E + E \Longrightarrow E + E + E \Longrightarrow y + E + E \Longrightarrow y + y + E \Longrightarrow y + y + y$ and $E \Longrightarrow E + E \Longrightarrow y + E \Longrightarrow y + E + E \Longrightarrow y + y + E \Longrightarrow y + y + y$.

Proposition (asserted but not proved in the text): The "ETF" grammar for expressions is unambiguous.

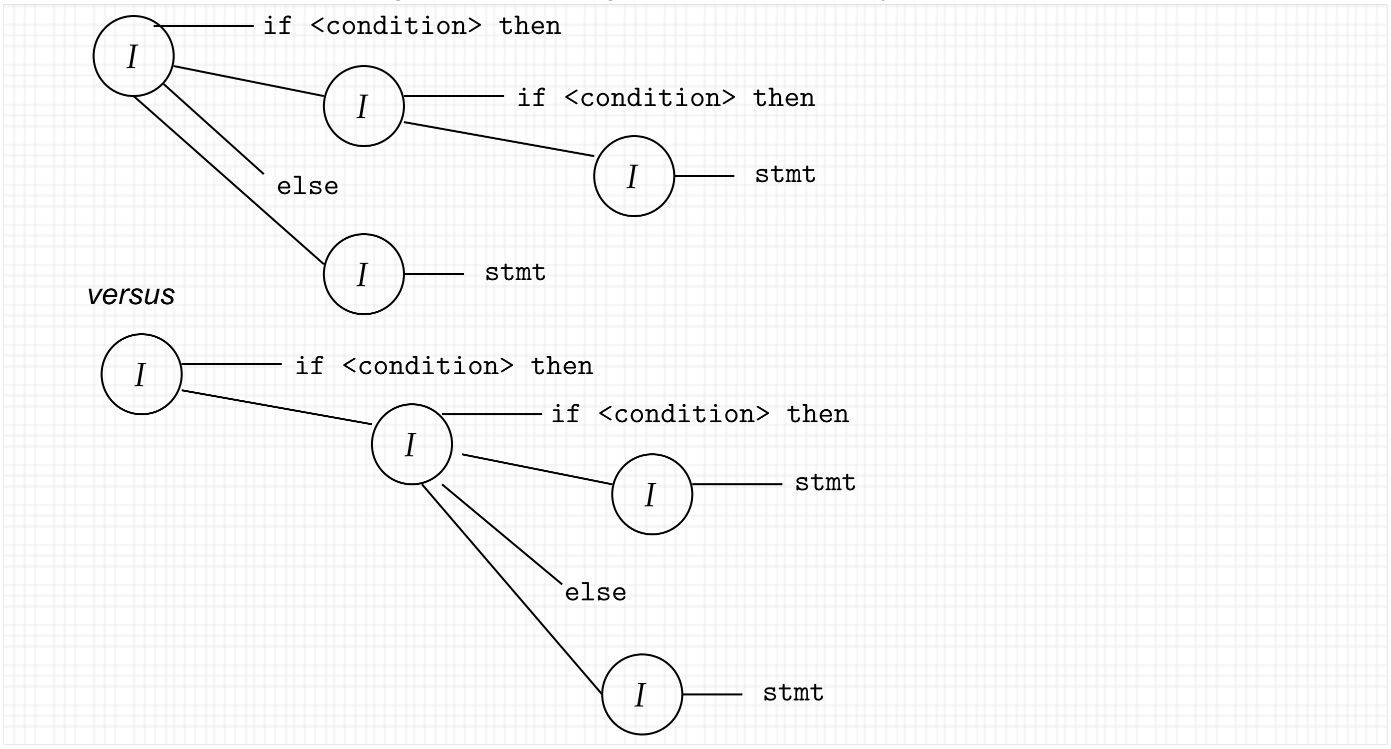
Example: In $I \rightarrow \epsilon \mid \$I \mid \$IdI$, when you have the string $\$\$d$, "which spear killed the dragon?"

$I \Longrightarrow \$IdI \Longrightarrow \$\$IdI \Longrightarrow \$\$dI \Longrightarrow \$\$d$. "First spear killed the dragon."
$I \Longrightarrow \$I \Longrightarrow \$\$IdI \Longrightarrow \$\$dI \Longrightarrow \$\$d$. "Second spear killed the dragon."

Now read I as `<statement>` (that is, a general statement, which could be compound such as an if-statement), $\$$ as `if <condition> then`, and d as `else`. Also read $I \rightarrow \epsilon$ as saying that the body represented by I becomes a basic statement, like an assignment statement. Then $\$\$d$ reads as:

`if <condition> then if <condition> then` (basic startement); `else` (basic statement);

Which `if` does the `else` part go with? Turning parse trees sideways to imitate indentation:

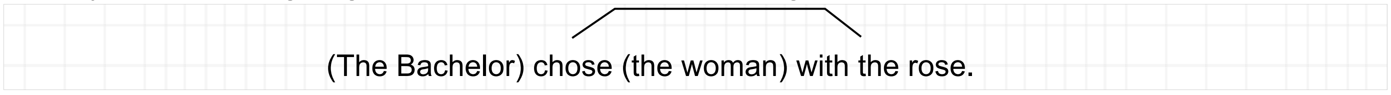


This ambiguity is *tolerated* by taking the second of these as the official reading: the dangling "else" associates with the inner "if".

Ambiguity occurs all the time in English and other human languages. There, contextual cues as to intended meaning often supply the disambiguation. Here is a variation on a notorious example in the text where the context might come out different from your expectation:

The Bachelor chose the woman with the rose.

You might parse this as (the bachelor) (chose) (the woman with the rose). But if you've watched the TV show, you know that giving a rose is the method of choosing. So the intended parse is:


(The Bachelor) chose (the woman) with the rose.