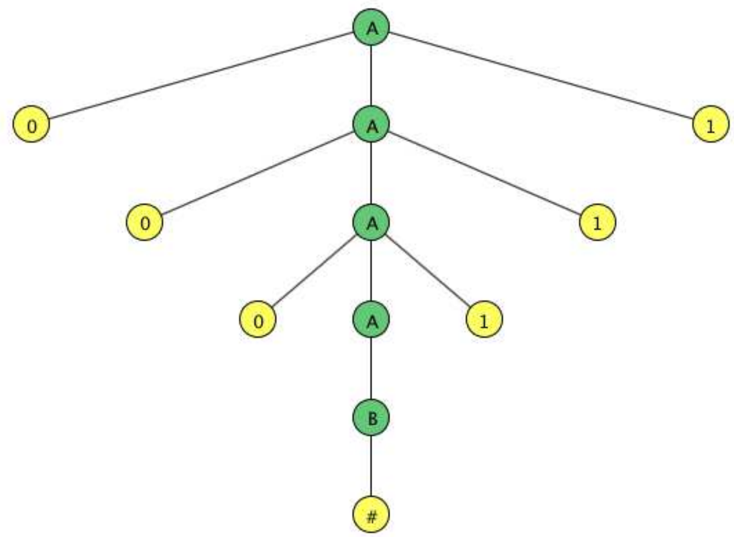
Parse Trees

- If $w \in L(G)$, for some CFG, then w has a parse tree representing the syntactical structure of w.

$$A \to 0A1 \mid B$$
$$B \to \#$$

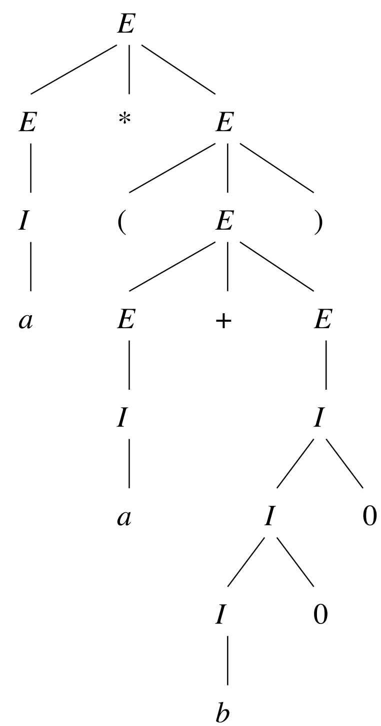


- Like derivations, a parse tree is an alternative representation for verifying if a string is in the language defined by a CFG.
- The process used for constructing a parse tree for a given string is similar to a derivation of the string.

Constructing a parse tree

We are particularly interested in parse trees where: the yield (string of leafs, from left to right) is a terminal string, and the root is the start symbol.

$$E \to I \mid E + E \mid E * E \mid (E)$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$



The yield of the tree is $a * (a + b00)$.

Inference, Derivations, and Parse Trees

- Let $G = (V, \Sigma, R, S)$ be a CFG, and $A \in V$.
- The following methods to determine if w is in the language of A are equivalent:
  - Derivations: $A \overset{*}{\Rightarrow} w$, $A \underset{lm}{\overset{*}{\Rightarrow}} w$, or $A \underset{rm}{\overset{*}{\Rightarrow}} w$
  - Construction of the parse tree with root A and yield w.

Exercise

- Provide a grammar for the language $0^*1^n0^n1^*, n > 1$.
- Provide a parse tree for the following string:

$$0110011$$

Ambiguity in Grammars and Languages

- A grammar captures the structure of a string through the parse tree.
- But is this structure always unique?
- Depending on the application, uniqueness of the structure is highly desirable, e.g., compilers and translators.
- A grammar is ambiguous if a string in the language has two different leftmost (or rightmost) derivations or parse trees.

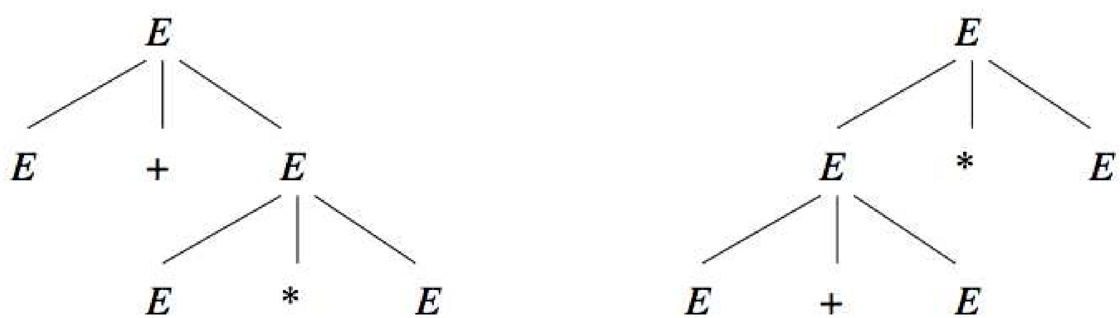
Let's see an example.

Ambiguity in Grammars

- A grammar $G = (V, \Sigma, R, S)$ is ambiguous if there is a string in Σ^* that has more than one parse tree.
- Example: In the grammar: $E \to I \mid E + E \mid E * E \mid (E) \cdots$
  The sentential form $E + E * E$ has two different parse trees:

$$E \Rightarrow E + E \Rightarrow E + E * E$$

$$E \Rightarrow E * E \Rightarrow E + E * E$$



Removing ambiguity from grammars

- Good news: there are ad hoc methods to reduce and remove ambiguity
- Bad news: there is no general algorithm to remove umbiguity. Worse yet: some grammars are inherently ambiguous.

Let’s study some techniques for spotting and reducing ambiguity in grammars.

Techniques for reducing ambiguity

We'll consider three grammar structures that often lead to ambiguity:

- ϵ rules like $S \to \epsilon$
- Symmetric recursive rules like $S \to SS$
- Rules that lead to ambiguous attachment of optional postfixes, e.g., $S \to aS|aSb$

Why ϵ rules are problematic? Consider $S \to SS|a|\epsilon$.

- There are many possible derivations/parse trees for the string a as we can use $S \to SS$ repeatedly, and then get rid of the unnecessary Ss by using $S \to \epsilon$. E.g.:

$$S \Rightarrow SS \Rightarrow SSS \Rightarrow aSS \Rightarrow aS \Rightarrow a$$

$$S \Rightarrow SS \Rightarrow S \Rightarrow a$$

Eliminating ϵ-rules

- *Basic idea*: Suppose A is nullable (i.e., $A \stackrel{*}{\Rightarrow} \epsilon$). We'll then replace a rule like like $C \rightarrow BAD$ with $C \rightarrow BAD, C \rightarrow BD$ and delete any rules with body ϵ.

Algorithm $RemoveEps(G)$, where $G = (V, T, R, S)$:

1. *Obtain the set of all nullable symbols,* $n(G)$, *in* G:
   - **Basis**: For all rules $A \rightarrow \epsilon \in R$, include A in $n(G)$.
   - **Induction**: For all rules $A \rightarrow C_1 C_2 \cdots C_k \in R$. If $\{C_1, C_2, \cdots, C_k\} \subseteq n(G)$, then include A in $n(G)$.
2. *Obtain the new grammar* G_1: for each rule $A \rightarrow X_1 X_2 \cdots X_k$ of R, suppose m of the k X_i's s are nullable. Then G_1 will contain 2^m versions of this rule, where the nullable X_i's in all combinations are present or absent.

Eliminating ϵ-rules: example

- Let G be $S \to AB, A \to aAA \mid \epsilon, B \to bBB \mid \epsilon$
- Now $n(G) = \{A, B, S\}$. The first rule will become: $S \to AB \mid A \mid B$, the second $A \to aAA \mid aA \mid aA \mid a$, and the third $B \to bBB \mid bB \mid bB \mid b$
- We then delete the redundant rules, and end up with grammar G_1 : $S \to AB \mid A \mid B, A \to aAA \mid aA \mid a, B \to bBB \mid bB \mid b$

Ok, I got it. But what if L(G) contains ϵ and it is important to retain it? E.g.:

$$S \to (S)$$

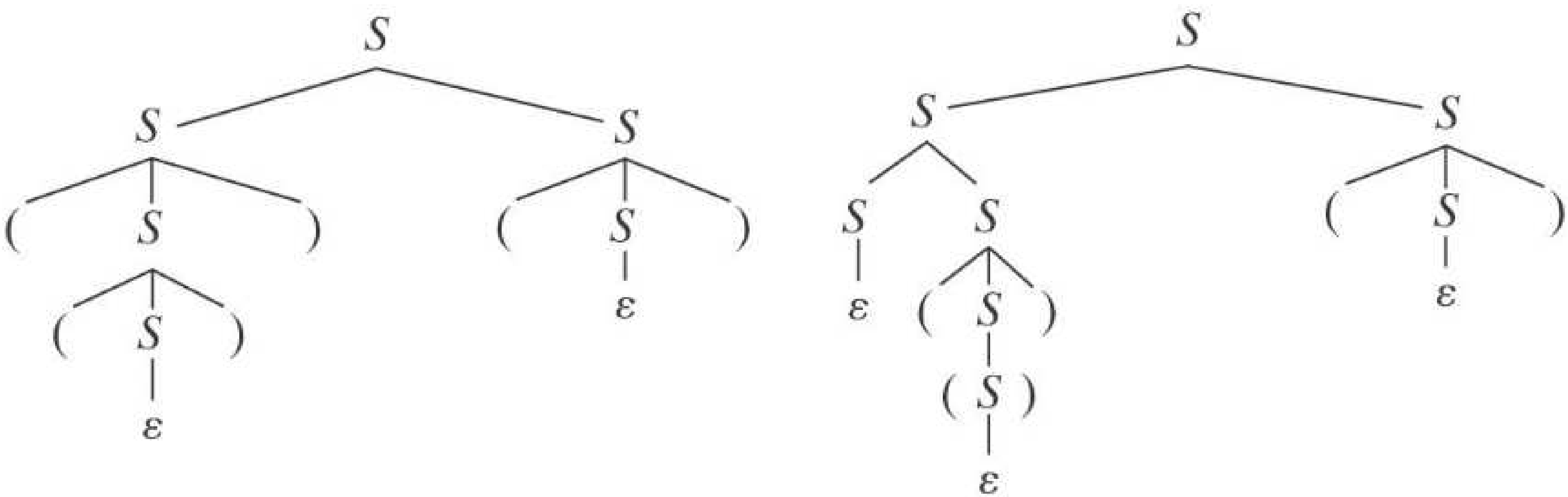
$$S \to SS$$

$$S \to \epsilon$$

A highly ambiguous grammar

$$S \to (S)$$
$$S \to SS$$
$$S \to \epsilon$$

Parse tree for the string $(())()$



When $\epsilon \in L(G)$

We use the following algorithm to rewrite the grammar.

$atMostOneEps(G)$:

1. $G'' = RemoveEps(G)$.
2. If start symbol S of G is nullable then
   (a) Create in G'' a new start symbol S'.
   (b) Add to R the two rules:

$$S' \to \epsilon$$
$$S' \to S$$

   (c) Return G''.

Applying $atMostOneEps(G)$

Original Grammar:

$$S \to (S)$$
$$S \to SS$$
$$S \to \epsilon$$

$removeEps$**:**

$$S \to (S)$$
$$S \to ()$$
$$S \to SS$$

Result of $atMostOneEps$**:**

$$S' \to \epsilon$$
$$S' \to S$$
$$S \to (S)$$
$$S \to ()$$
$$S \to SS$$

But there is still ambiguity

$S' \to \epsilon$

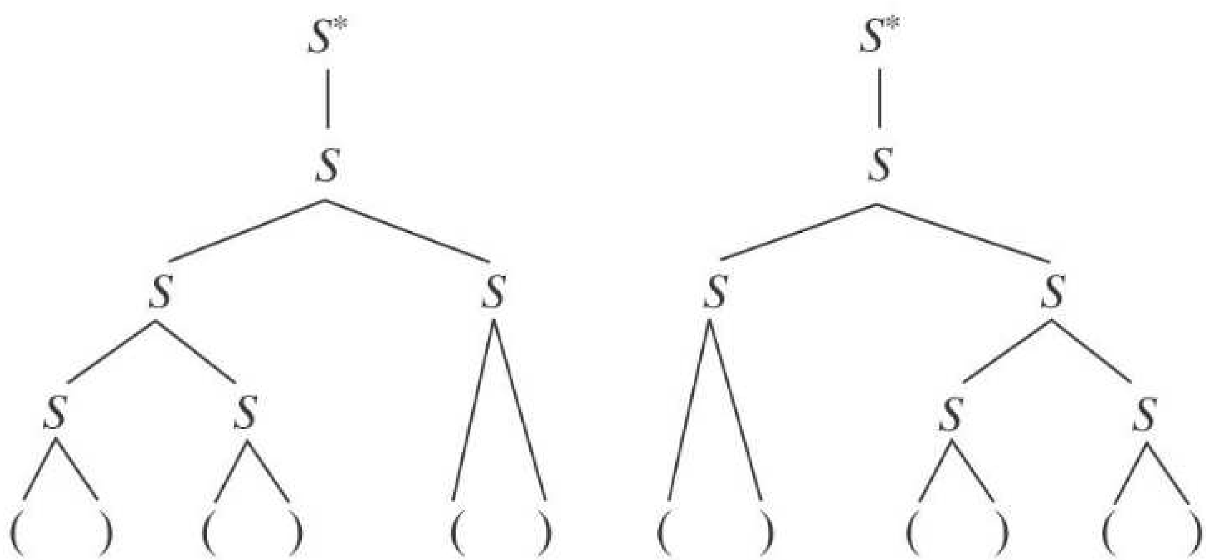
$S' \to S$

$S \to (S)$

$S \to ()$

$S \to SS$

What about $()()()$?



Eliminating symmetric recursive rules

$$S' \to \epsilon$$
$$S' \to S$$
$$S \to (S)$$
$$S \to ()$$
$$S \to SS$$

- Replace $S \to SS$ with one of:

  $S \to SS_1$ force branching to the left
  $S \to S_1S$ force branching to the right

- add $S \to S_1$ to the grammar, and
- change $S \to (S), S \to ()$ to $S_1 \to (S), S_1 \to ()$

So we get

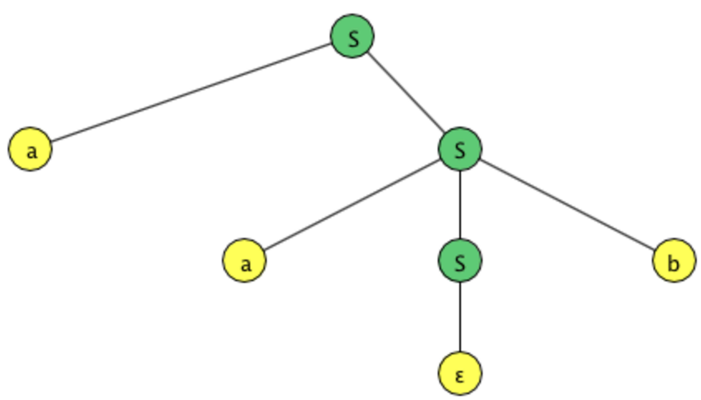
$S' \to \epsilon$	$S \to SS_1$
$S' \to S$	$S \to S_1$
$S_1 \to (S)$	
$S_1 \to ()$	

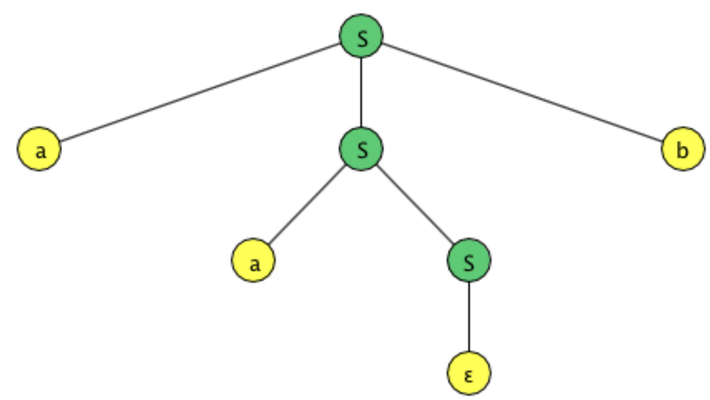
Ambiguous attachment

A third source for ambiguity arises when constructs with optional fragments are nested. E.g.

$$S \rightarrow aS \mid aSb$$

Two different parse trees for the string `aab`





Exercise: provide a unambiguous grammar that recognizes the same language.

Ambiguous attachment (cont.)

The dangling else problem:

```
<stmt> ::= if <cond> then <stmt>
<stmt> ::= if <cond> then <stmt> else <stmt>
```

Consider:

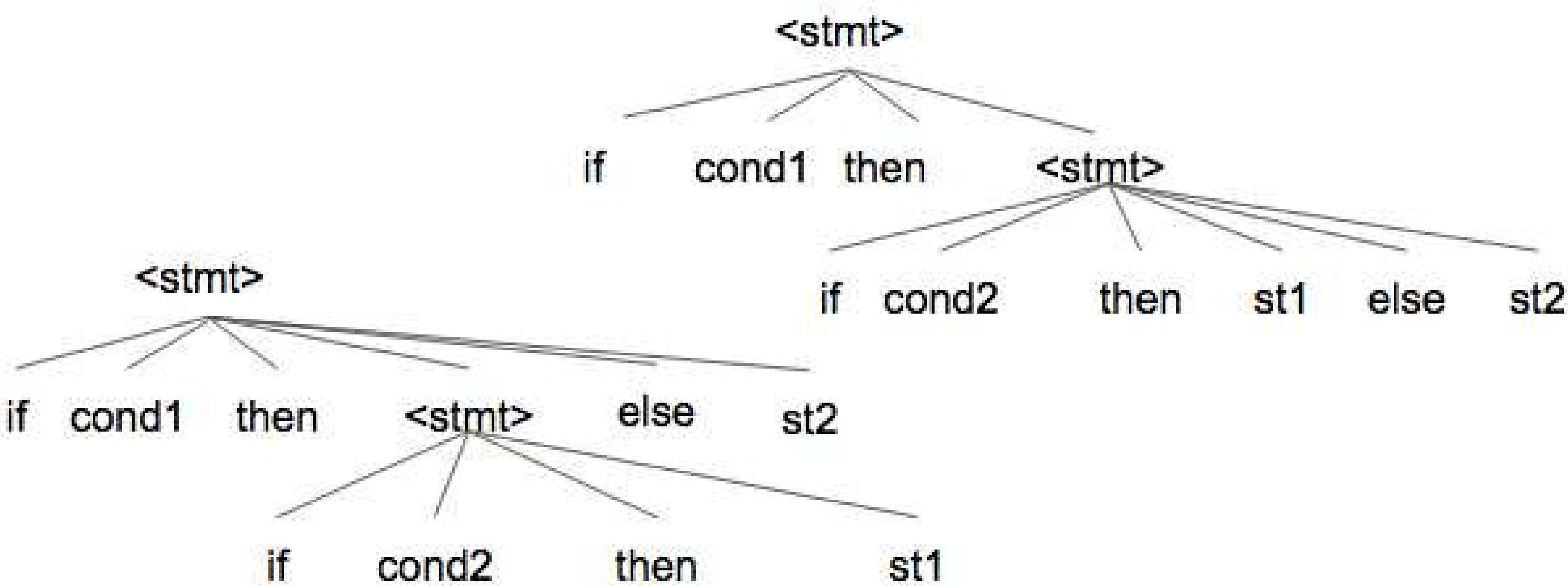
`if` $cond_1$ `then if` $cond_2$ `then` st_1 `else` st_2

Should the `else` go with with the innermost `if` or with the outermost `if`?

The dangling else problem

`<stmt> ::= if <cond> then <stmt>`
`<stmt> ::= if <cond> then <stmt> else <stmt>`

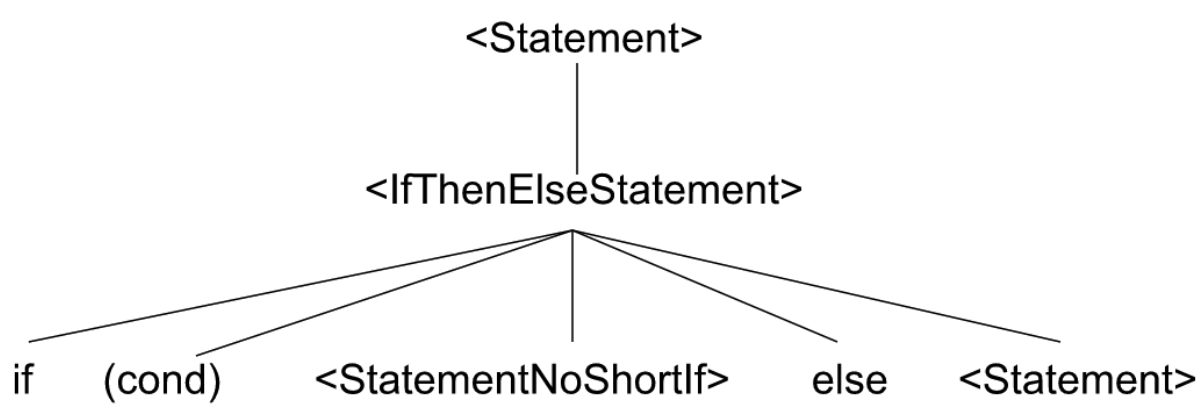
if $cond_1$ then if $cond_2$ then st_1 else st_2



The Java Fix

The grammar guarantees that, if a top-level `if` has an `else`, then the embedded `if` must also have one.

<Statement> ::= <IfThenStatement> | <IfThenElseStatement> |
<IfThenElseStatementNoShortIf>
<StatementNoShortIf> ::= <block> |
<IfThenElseStatementNoShortIf> | …
<IfThenStatement> ::= `if` (<Expression>) <Statement>
<IfThenElseStatement> ::= `if` (<Expression>)
<StatementNoShortIf> `else` <Statement>
<IfThenElseStatementNoShortIf> ::=
`if` (<Expression>) <StatementNoShortIf>
`else` <StatementNoShortIf>



Arithmetic Expressions: a better way

Let's study this grammar:

$$E \to I \mid E + E \mid E * E \mid (E)$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

- Problems: no precedence between * and +, and no left (or right) associativity.
- How to solve that? Redesign the grammar so that parse trees would reflect such structure.

Solution

Introducing variables that represent "binding strength".

1. A factor is an expression that cannot be broken apart by an adjacent $*$ or $+$. E.g.: Identifiers and a parenthesized expression. $F \to I \mid (E)$
2. A term is an expression that cannot be broken by $+$. E.g.: $a * b$, or a factor. $T \to F \mid T * F$
3. The rest are expressions, i.e., they can be broken apart with $*$ or $+$. $E \to T \mid E + T$

The original grammar:

$$
\begin{aligned}
E &\to I \mid E + E \mid E * E \mid (E) \\
I &\to a \mid b \mid Ia \mid Ib \mid I0 \mid I1
\end{aligned}
$$

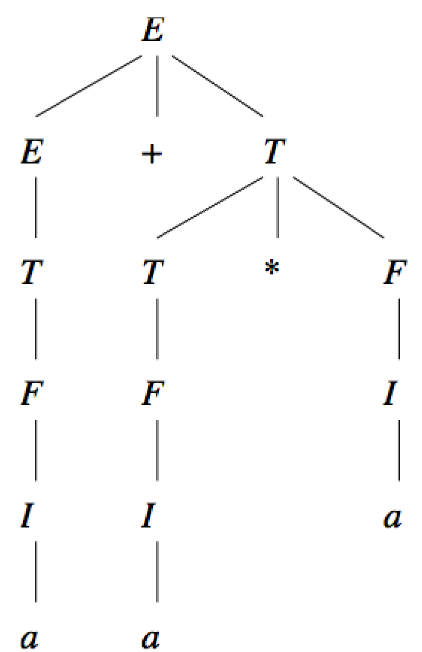
The redesigned grammar:

$$
\begin{aligned}
E &\to T \mid E + T \\
T &\to F \mid T * F \\
F &\to I \mid (E) \\
I &\to a \mid b \mid Ia \mid Ib \mid I0 \mid I1
\end{aligned}
$$

The redesigned grammar

$$E \to T \mid E + T$$
$$T \to F \mid T * F$$
$$F \to I \mid (E)$$
$$I \to a \mid b \mid Ia \mid Ib \mid I0 \mid I1$$

Now the only parse tree for $a + a * a$ will be



Exercise

Suppose you are designing a programming language and want to specify the syntax for valid type declarations, e.g.:

```
int x, z=3;
real y;
complex s;
```

Provide a grammar that defines the syntax for such type declarations.
Assumptions:

- variable identifiers: x, y, z, s
- numbers: $0, 1, 2, \cdots, 9$
- types: $int, real, complex$