Some more sums

Grammar

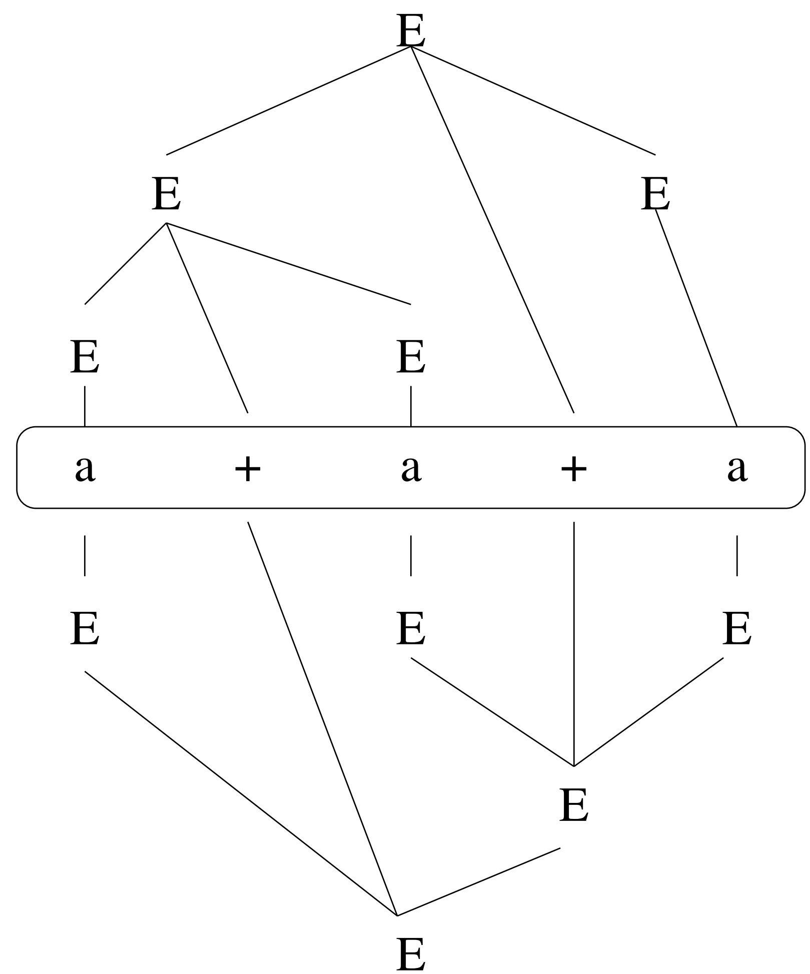
$$\begin{array}{lcl} \mathbf{E} & \rightarrow & \mathbf{E + E} \\ & | & \mathbf{a} \end{array}$$

Leftmost derivation

$$\begin{array}{lcl} \mathbf{E} & \Rightarrow & \boxed{\mathbf{E}}\mathbf{+E} \\ & \Rightarrow & \boxed{\mathbf{E+E}}\mathbf{+E} \\ & \Rightarrow & \mathbf{a+E+E} \\ & \Rightarrow & \mathbf{a+a+E} \\ & \Rightarrow & \mathbf{a+a+a} \end{array}$$

Another leftmost derivation

$$\begin{array}{lcl} \mathbf{E} & \Rightarrow & \boxed{\mathbf{E}}\mathbf{+E} \\ & \Rightarrow & \boxed{\mathbf{a}}\mathbf{+E} \\ & \Rightarrow & \mathbf{a+E+E} \\ & \Rightarrow & \mathbf{a+a+E} \\ & \Rightarrow & \mathbf{a+a+a} \end{array}$$

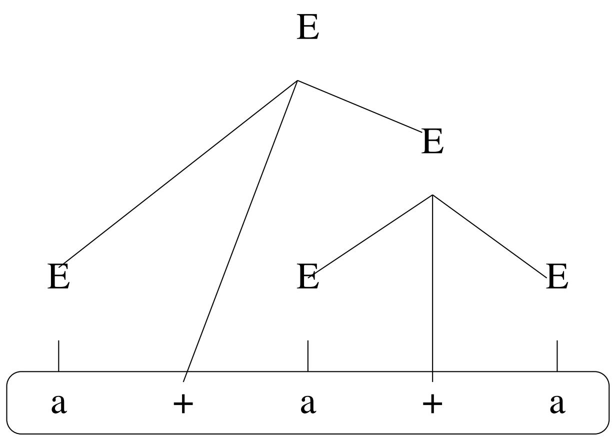


If the same string has two parse trees by a grammar G, then G is *ambiguous*. Equivalently, there are two distinct leftmost derivations of some string. Note that the language above is *regular*.

Ambiguity

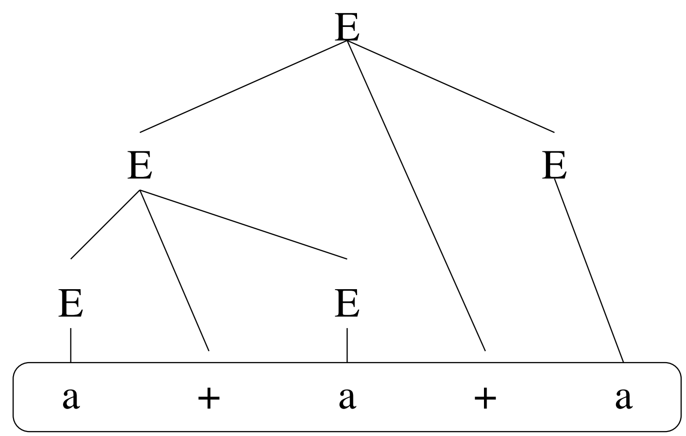
The parse tree below structures the input string as

$$(a + (a + a))$$



The parse tree below structures the input string as

$$((a + a) + a)$$



- With addition, the two expressions may be semantically the same. What if the a's were the operands of subtraction?
- How could a compiler choose between multiple parse trees for a given string?
- Unfortunately, there is (provably) no mechanical procedure for determining if a grammar is ambiguous; this is a job for human intelligence. However, compiler construction tools such as YACC can greatly facilitate the location and resolution of grammar ambiguities.
- It's important to emphasize the difference between a *grammar* being ambiguous, and a *language* being (inherently) ambiguous. In the former case, a different grammar may resolve the ambiguity; in the latter case, there exists no unambiguous grammar for the language.

Syntactic ambiguity

A great source of humor in the English language arises from our ability to construct interesting syntactically ambiguous phrases:

1. **I fed the elephant in my tennis shoes.**
   What does "in my tennis shoes" modify?
   (a) Was I wearing my tennis shoes while feeding the elephant?
   (b) Was the elephant wearing or inside my tennis shoes?
2. **The purple people eater.** What is purple?
   (a) Is the eater purple?
   (b) Are the people purple?

Suppose we modified the grammar for C, **so that any** {...} **block could be treated as a primary value.**

```
{ int i; i=3*5; } + 27;
```

would seem to have the value 42. But if we just rearrange the white space, we can get

```
{int i; i=3*5; }
+27;
```

which represents two statements, the second of which begins with a unary plus.

A good assignment along these lines is to modify the C grammar to allow this simple language extension, and ask the students to determine what went wrong. The students should be fairly comfortable using YACC before trying this experiment.

Semantic ambiguity

In English, we can construct sentences that have only one parse, but still have two different meanings:

1. **Milk drinkers turn to powder.** Are more milk drinkers using powdered milk, or are milk drinkers rapidly dehydrating?
2. **I cannot recommend this student too highly.** Do words of praise escape me, or am I unable to offer my support.

In programming languages, the language standard must make the meaning of such phrases clear, often by applying elements of context.

For example, the expression

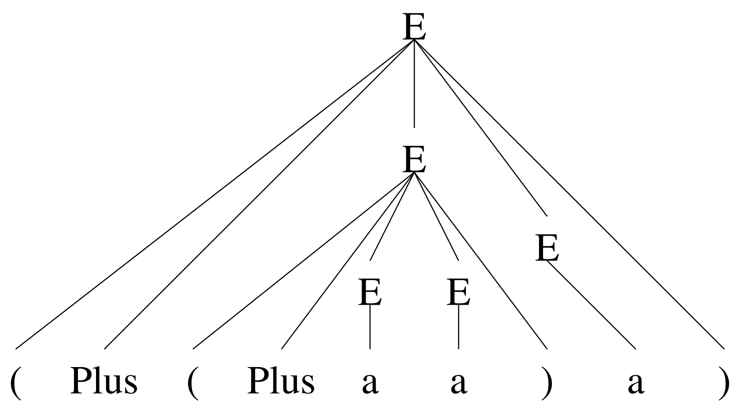
$$a + b$$

could connote an integer or floating-point sum, depending on the types of a and b.

A nonambiguous grammar

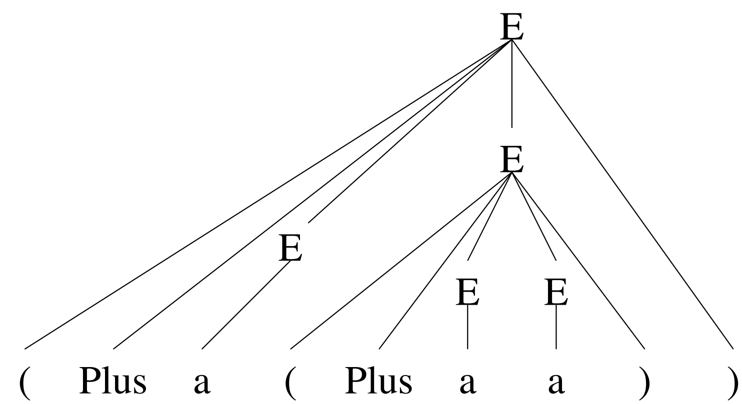
$$
\begin{array}{lll}
\mathbf{E} & \rightarrow & \textbf{(Plus E E)} \\
 & \mid & \textbf{(Minus E E)} \\
 & \mid & \textbf{a}
\end{array}
$$

It's interesting to note that the above grammar, intended to generate LISP-like expressions, is not ambiguous.



is the prefix equivalent of

$$((a + a) + a)$$



is the prefix equivalent of

$$(a + (a + a))$$

These are two *different strings* from this language, each associated explicitly with a particular grouping of the terms. Essentially, the parentheses are syntactic sentinels that simplify construction of an unambiguous grammar for this language.

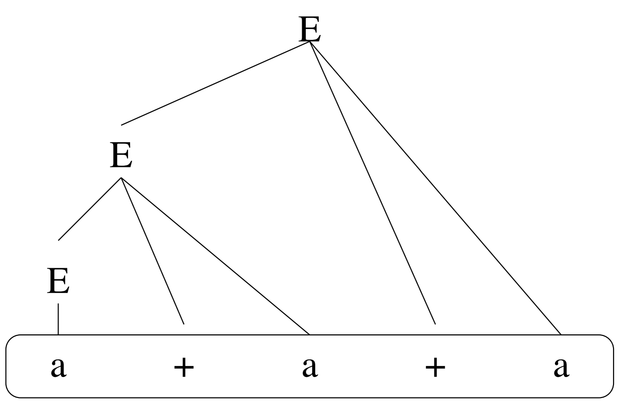
Addressing ambiguity

$$\begin{array}{lcl} E & \rightarrow & E + E \\ & | & a \end{array}$$

We'll try to rewrite the above grammar, so that in a (leftmost) derivation, there's only one rule choice that derives longer strings.

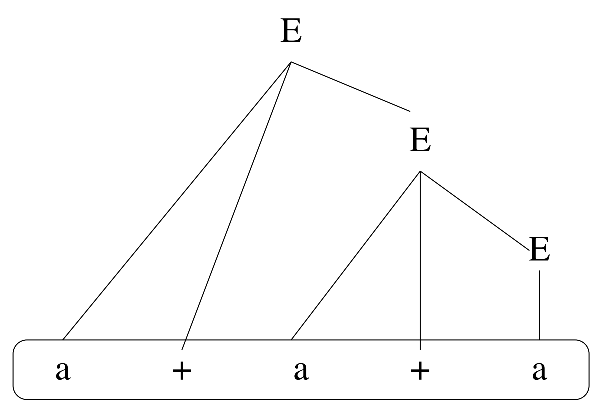
$$\begin{array}{lcl} E & \rightarrow & E + a \\ & | & E - a \\ & | & a \end{array}$$

These rules are *left recursive*, and the resulting derivations tend to associate operations from the left:



$$\begin{array}{lcl} E & \rightarrow & a + E \\ & | & a - E \\ & | & a \end{array}$$

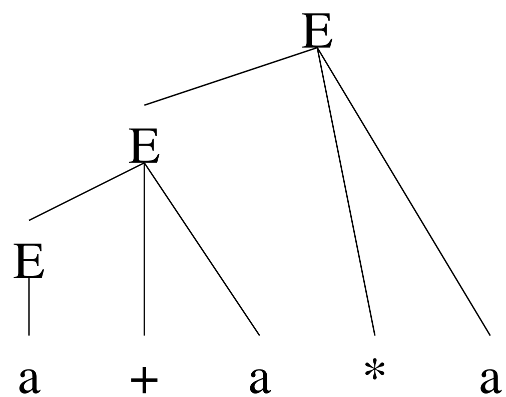
The grammar is still unambiguous, but strings are now associated from the right:



Addressing ambiguity (cont'd)

Our first try to expand our grammar might be:

$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{E + a} \\ & | & \textbf{E} * \textbf{a} \\ & | & \textbf{a} \end{array}$$



The above parse tree does not reflect the usual precedence of $*$ over $+$.

To obtain *sums of products*, we revise our grammar:

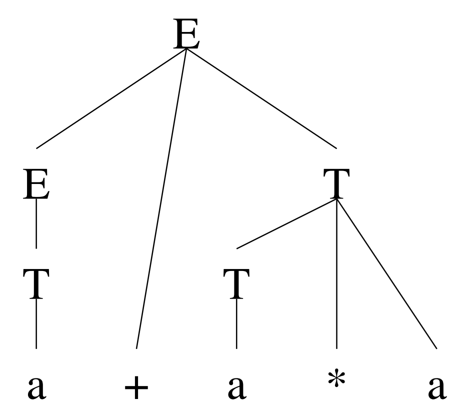
$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{E + T} \\ & | & \textbf{T} \end{array}$$

This generates strings of the form

$$T + T + \ldots + T$$

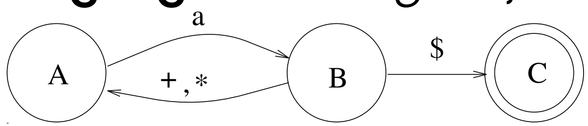
We now allow each T to generate strings of the form $a * a * \ldots * a$

$$\begin{array}{lcl} \textbf{E} & \rightarrow & \textbf{E + T} \\ & | & \textbf{T} \\ \textbf{T} & \rightarrow & \textbf{T} * \textbf{a} \\ & | & \textbf{a} \end{array}$$

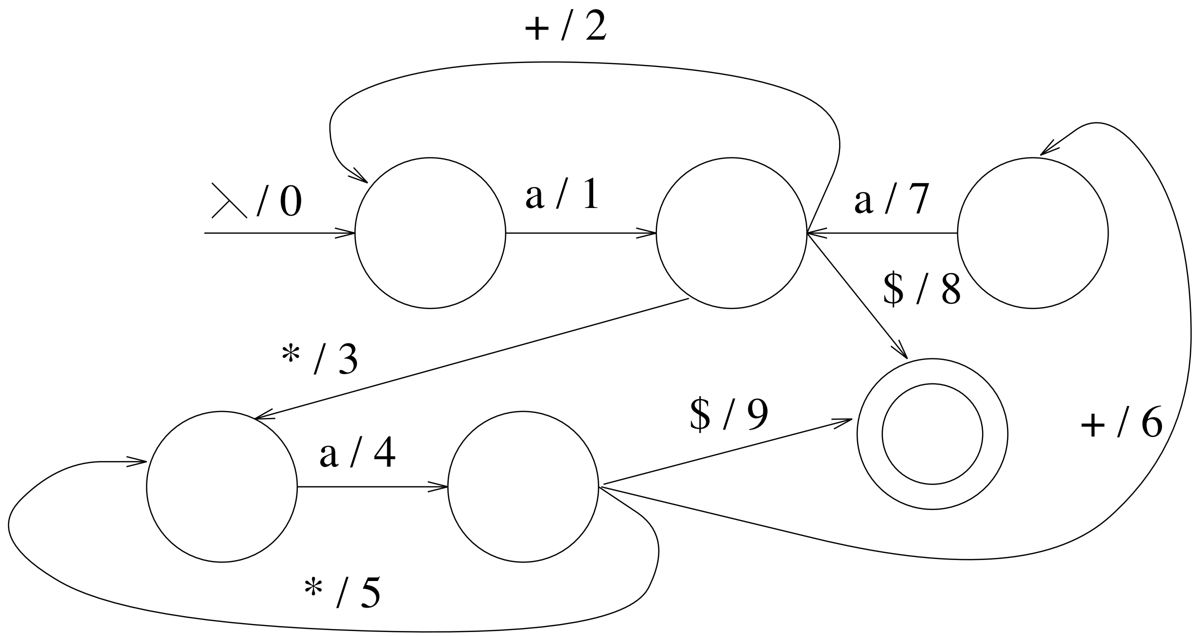


Translating two-level expressions

Since our language is still *regular*, a finite-state machine could do the job. While the machine



could do the job, there's not enough "structure" to this machine to accomplish the prioritization of $*$ over $+$. However, the machine below can do the job.



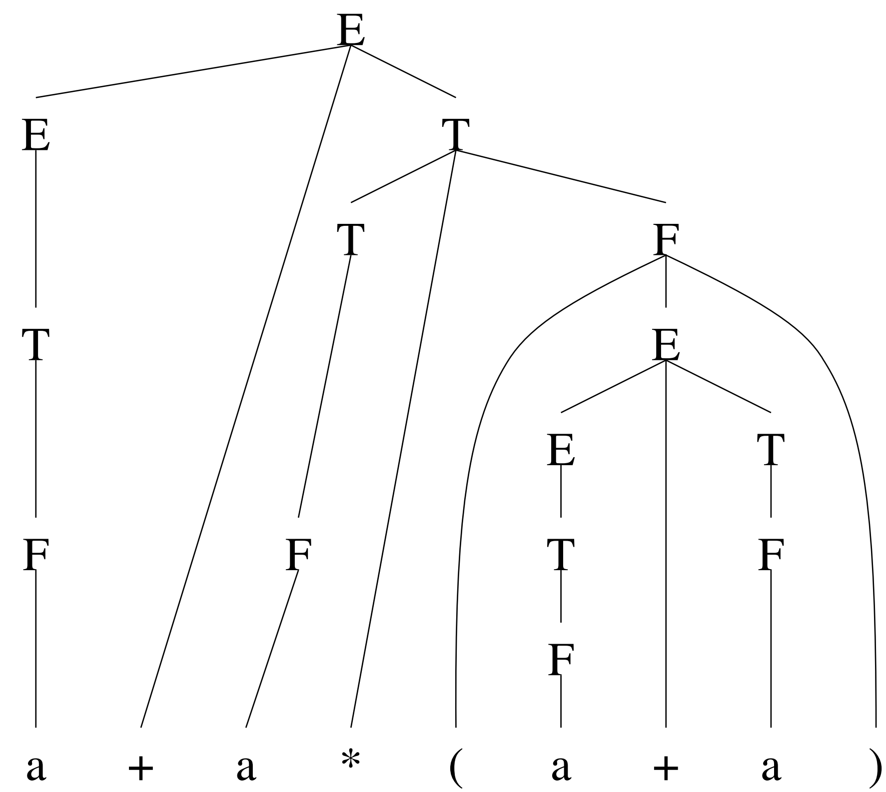
0	$Sum = 0$	5	$Prod = Prod \times Acc$
1	$Acc = a$	6	$Sum = Sum + (Prod \times Acc); \; Prod = 1$
2	$Sum = Sum + Acc$	7	$Acc = a$
3	$Prod = Prod \times Acc$	8	$Sum = Sum + Acc$
4	$Acc = a$	9	$Sum = Sum + (Prod \times Acc); \; Prod = 1$

Let's add parentheses

While our grammar currently structures inputs appropriately for operator priorities, parentheses are typically introduced to override default precedence. Since we want a parenthesized expression to be treated "atomically", we now generate sums of products of parenthesized expressions.

$$
\begin{array}{lcl}
\textbf{E} & \rightarrow & \textbf{E + T} \\
 & | & \textbf{T} \\
\textbf{T} & \rightarrow & \textbf{T} * \textbf{F} \\
 & | & \textbf{F} \\
\textbf{F} & \rightarrow & \textbf{(E)} \\
 & | & \textbf{a}
\end{array}
$$

This grammar generates a nonregular language. Therefore, we need a more sophisticated "machine" to parse and translate its generated strings.



The grammar we have developed thus far is the textbook "expression grammar". Of course, we should make a into a nonterminal that can generate identifiers, constants, procedure calls, *etc*.