CSE305: Programming Languages Week 3

Grammars, Derivations, and Ambiguity

First, a blast of formal definitions that the text stops short of giving:

Definition: A **context-free grammar** (**in BNF notation**) has a set V of **nonterminals** (also called grammar variables), a set T of **terminals** (that is disjoint from V), and a set R of rules of the form

$$A \ ::= \ X$$

where A is a single nonterminal and X is a string of terminals and nonterminals. If X is just a single nonterminal (or single char or token), then $A \ ::= \ X$ is a **unit rule**. In **extended BNF** (**EBNF**, but often just called BNF), X may include the "EBNF metachar" constructs

- $[Y]$ meaning that the Y part is optional; some notations write Y_{opt} instead.
- $\{Y\}$ meaning that the Y part may be used 0 or more times (which includes its being optional); some EBNF notations write $\{Y\}^*$ for this.
- $\{Y\}^+$ meaning Y may be used 1 or more times (so not optional), and
- $(Y \mid Z)$ meaning either Y or Z may be used---must use one and can't use both.

Definition: A **derivation** in a BNF grammar is a sequence

$$A \implies Z_1 \implies Z_2 \implies \cdots \implies Z_k$$

where $Z_i \implies Z_j$ means that there is a nonterminal A inside Z_i and a rule $A \ ::= \ X$ such that substituting X for A inside Z_i gives exactly Z_j. If X has EBNF metachars, then first resolve them to say exactly what conforming rule $A \ ::= \ X'$ is actually used, and then substitute X' for A. We say that A **derives** Z_k in k **steps**. By convention, A derives itself in 0 steps, so we can write $A \implies^* Z$ to mean that A derives Z in some number (i.e., zero or more) of steps.

If A is considered the start symbol of the grammar (typically the syntactic category "compilation unit" for a whole programmijng language, but can be "EXP" or "TYPE" or "PAT" for what we're focusing on now), then each Z_i can be called a **sentiential form**. If Z_k is all terminals, it is the **yield** of the derivation.

Definition: A **parse tree** T is a tree, each of whose internal nodes (i.e., non-leaf nodes) is labeled with a single nonterminal A, and whose children are labeled with the individual chars (or tokens) of a rule $A \ ::= \ X$. The **yield** of T is the sequence of terminals in its leaves reading left-to-right.

Note that a subtree of a parse tree from any internal node is also a parse tree, whose yield is a substring of the overall tree's yield. Various conventions can be applied, such as shortcutting chains of unit-rule applications like EXP ⟹ <variable> ⟹ foo into a single step, or leaving some nonterminals unexpanded, treating them as if they were leaves.

Definition: A derivation is **leftmost** if in every step $Z_i \implies Z_j$, the leftmost nonterminal in Z_i is expanded. It is **rightmost** if in every step, the rightmost nonterminal gets expanded.

Fact: Every derivation builds a unique parse tree, but multiple derivations can build the same parse tree. However, every parse tree gives a unique leftmost derivation by doing a left-to-right **preorder transversal** of the tree. It also gives a unique rightmost derivation by doing the transversal right-to-left instead.

Definition: A terminal string (or more generally, a sentential form) Z is **ambiguous** with respect to a given grammar if it has two or more different parse trees via that grammar. This is equivalent to Z having two or more different leftmost derivations, and to having two or more different rightmost derivations. Otherwise, it is **unambiguous** for the grammar.

A string Z may be ambiguous in one grammar but unambiguous in others. The idealized goal is:

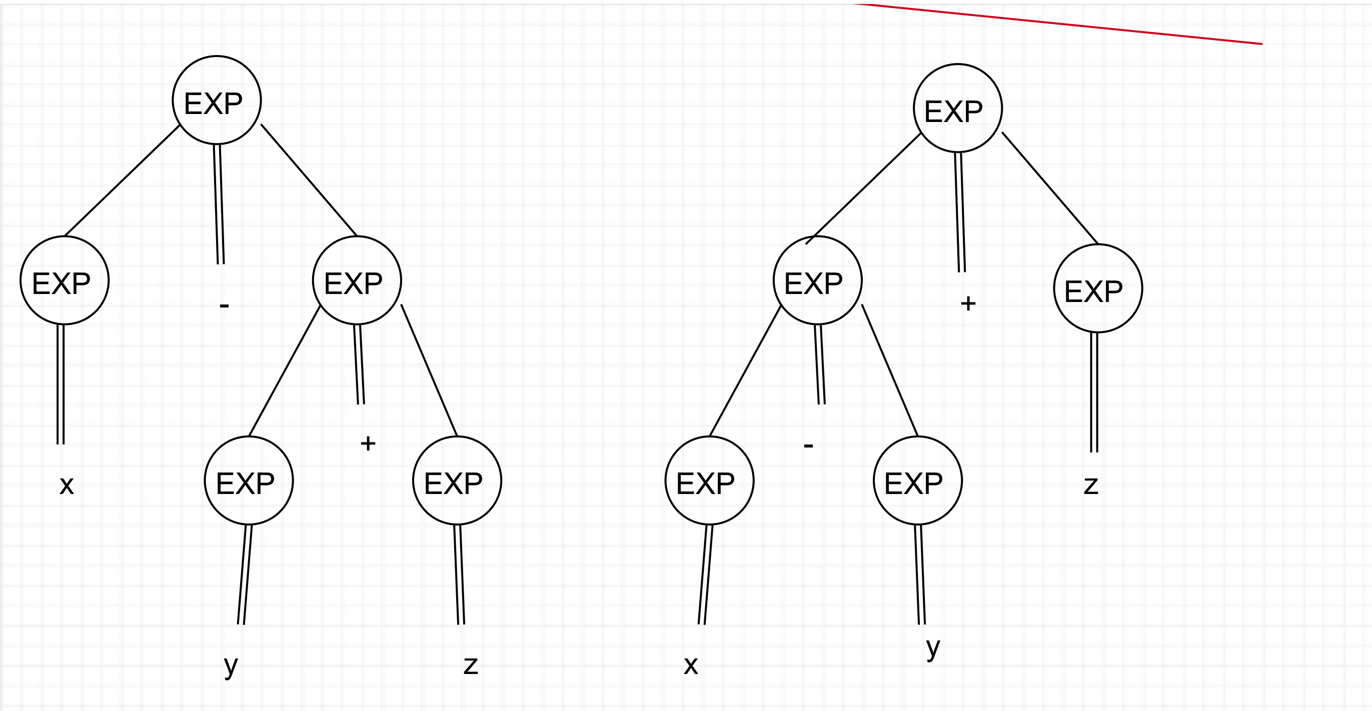
Definition: A grammar is **unambiguous** if *every* terminal string it yields is unambiguous in that grammar.

Ambiguity and How It Can Mislead

Let's first revisit our simple expression grammar:

EXP ::= <constant> | <variable> | - EXP | EXP <binop> EXP   <binop> ::= + | - | * | /

Consider the terminal string **`x - y + z`**. Here are two different parse trees and their corresponding leftmost derivations:



EXP ⟹ EXP - EXP ⟹ x - EXP ⟹ x - EXP + EXP ⟹ x - y + EXP ⟹ x - y + z
EXP ⟹ EXP + EXP ⟹ EXP - EXP + EXP ⟹ x - EXP + EXP ⟹ x - y + EXP ⟹ x - y + z

The ambiguity is palpable. If, say, $x = 8$ and $y = 3$ and $z = 4$, the former parse groups as $8 - (3 + 4) = 1$, whereas the second intuitively does $(8 - 3) + 4 = 9$. Which should it be?

[Class also did rightmost derivations:
EXP ⟹ EXP - EXP ⟹ EXP - EXP + EXP ⟹ EXP - EXP + z ⟹ EXP - y + z ⟹ x - y + z
EXP ⟹ EXP + EXP = EXP + Z ⟹ EXP - EXP + z ⟹ EXP - y + z ⟹ x - y + z.]

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 famous Sipser theory-of-computation 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.

Here's another derivation that is even more "rogue", now writing just E for EXP:

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

Shouldn't we have grouped $y + z$? Well, we can provide the option to do so:

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

so we can derive

$$E \implies E{*}E \implies^2 x{*}E \implies x{*}(E) \implies x{*}(E + E) \implies^2 x{*}(y + E) \implies^2 x{*}(y + z).$$

But this doesn't solve the problewm of the original derivation being legal. AND the ambiguity of x*y+z shows up in the better-behaved derivation:

$$E \implies E + E \implies E{*}E \; + \; E \implies^6 x{*}y \; + \; z.$$

Well, we can outlaw it by making parentheses always required:

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

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

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

or rightmost

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

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.
- 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.

Ways to Fix Ambiguity

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

Example: Mathematically, (3+x)*(y-z) = 3*y - 3*z + x*y - x*z. But they are quite different as expressions. Here is a leftmosat derivation of the left-hand side (LHS) in the "ETF" grammar:

E ⟹ T ⟹ T*F ⟹ F*F ⟹ (E)*F ⟹ (E+T)*F ⟹ (T+T)*F ⟹ (F+T)*F ⟹ (3+T)*F ⟹ (3+F)*F ⟹ (3+x)*F ⟹ (3+x)*(E) $\Longrightarrow^*$ (3+x)*(y-z)

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

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

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

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

Note: You can also do $E \Longrightarrow T \Longrightarrow F \Longrightarrow (E) \Longrightarrow (E+T)$ 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

$$E \Longrightarrow E+T \Longrightarrow T+T \Longrightarrow T{*}F+T \Longrightarrow F{*}F+T \Longrightarrow^4 x{*}y+T \Longrightarrow^3 x{*}y + z$$

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 $(x{*}(y) + ((z)))$.

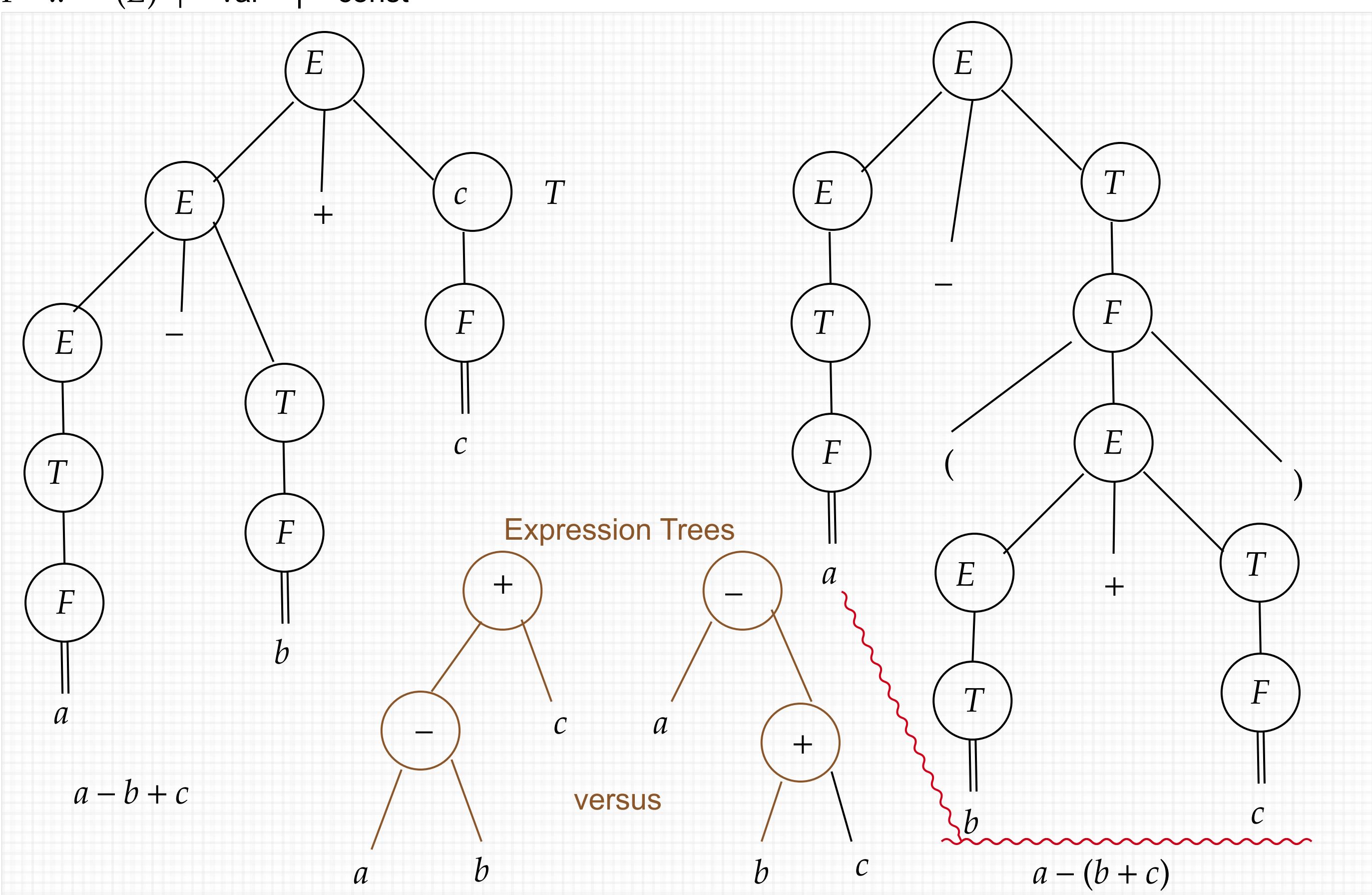
We can also tack on more syntactic categories, such as having a <factor> involve **powers**. Some programming languages have a native operation for powers like $**$, but you have to be careful that it is **right-associative**: $a{**}b{**}c$ means $a{**}(b{**}c) = a^{b^c}$, not $(a{**}b){**}c = \left(a^b\right)^c$ because the latter just becomes a^{bc}. The grammar would implement this by making F recurse *on the right* not left:

F ::= P | $P ** F$
P ::= (E) | <var> | <const>

Combining this with the rules for E and T like above creates what we could call an "**ETFP**"-type of grammar. When something like "P" is used, it is more often referred to as a "***primary expression***" than as a "power"---but the idea is similar.

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), the grammar also is instrumental to write the compiler's **parsing** stage.

Example: Here is how our previous ambiguous example $a - b + c$ works out with parse trees and leftmost derivations in the ETF grammar---we go back to the one without powering which is:
E ::= T | $E + T$ | $E - T$
T ::= F | $T * F$ | T / F
F ::= (E) | <var> | <const>



$E \Longrightarrow E + T \Longrightarrow E - T + T \Longrightarrow T - T + T \Longrightarrow F - T + T \Longrightarrow^{2} a - T + T \Longrightarrow^{6} a - b + c.$
$E \Longrightarrow E - T \Longrightarrow T - T \Longrightarrow F - T \Longrightarrow^{2} a - T \Longrightarrow a - F \Longrightarrow a - (E) \Longrightarrow a - (E + T)$
$\Longrightarrow a - (T + T) \Longrightarrow a - (F + T) \Longrightarrow^{2} a - (b + T) \Longrightarrow^{2} a - (b + c).$

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

Proposition (asserted but not proved in the text): The "ETF" grammar for expressions is unambiguous. So is the one with the added rule for powering.

This leads to a "design pattern" for unambiguous grammars that I call the "ski slope and chair lift" pattern. Here it is a grammar that is quote close to the official grammars for expressions in C/C++/Java etc."

```
E   ::=  E2 <assignment_op> E | E2          //assignment is right-associative
E2  ::=  E2 <binop> E3 | E3                 //binops are left-associative
E3  ::=  +E3 | -E3 | ++P | --P | E4         //pre-increment rules
E4  ::=  P++ | P-- | P                      //post-increment rules
P   ::=  (E) | <constant> | <variable>    (etc.)
<assignment_op> ::= = | += | -=    (etc.)
<binop>  ::=  == | != | + | - | * | /    (etc.)
```

Yes, assignment statements are classed as expressions that return values in these languages too. That is technically needed to support "multiple assignments" like `x = y = 3;` (though apart from speed-critical code, this is dubious). The grammar spells out the allowed unary operators rather than have a syntactic category for them. This includes separate lines for pre- and post- increment and decrement. Here is a simple challenge:

1. Can we derive a legal Java expression that has the substring "`++ + ++`" in it (noting the whitespace around the binary +)?

Yes: here is the derivation:

```
E ==> E2 ==> E2 BINOP E3 ==> E3 BINOP E3
  ==> E4 BINOP E3 ==> P++ BINOP E3 ==>^2 x++ BINOP E3 ==> x++ + E3
  ==> x++ + ++P ==>^2 x++ + ++y
```

The **main takeaway** is that the "**ETF**" or "**ski run (with chair lift)**" design pattern is so well entrenched, and manipulable with grammar **parser-generator** tools, that

1. giving simple grammar rules typified by `EXP ::= EXP <binop> EXP`,
2. stating **precedence** levels for the allowed operators, and
3. stating for each operator whether it **associates left** or **right**

is considered tantamount to giving an unambiguous ETF-style grammar. There are, however, other kinds of ambiguity that can't be dealt with unobtrusively. In fact, if you allow two different kinds of items A and B in a certain place, and $A \cap B$ is nonempty syntactically, then chances are you can't eradicate the ambiguity for any terminal syntax t in $A \cap B$. You can derive such a t from the rules for A or from the rules for B. I don't know whether so called **inherently ambiguous context-free languages** have actually cropped up in the design of any real programming language (they could be handled by the grammar add-on of **attributes** but the text doesn't go this far in a section of chapter 3 that is OK to skim). However, the common ambiguity we discuss next tends to be *tolerated* rather than rewritten.

The Dangling Else Ambiguity

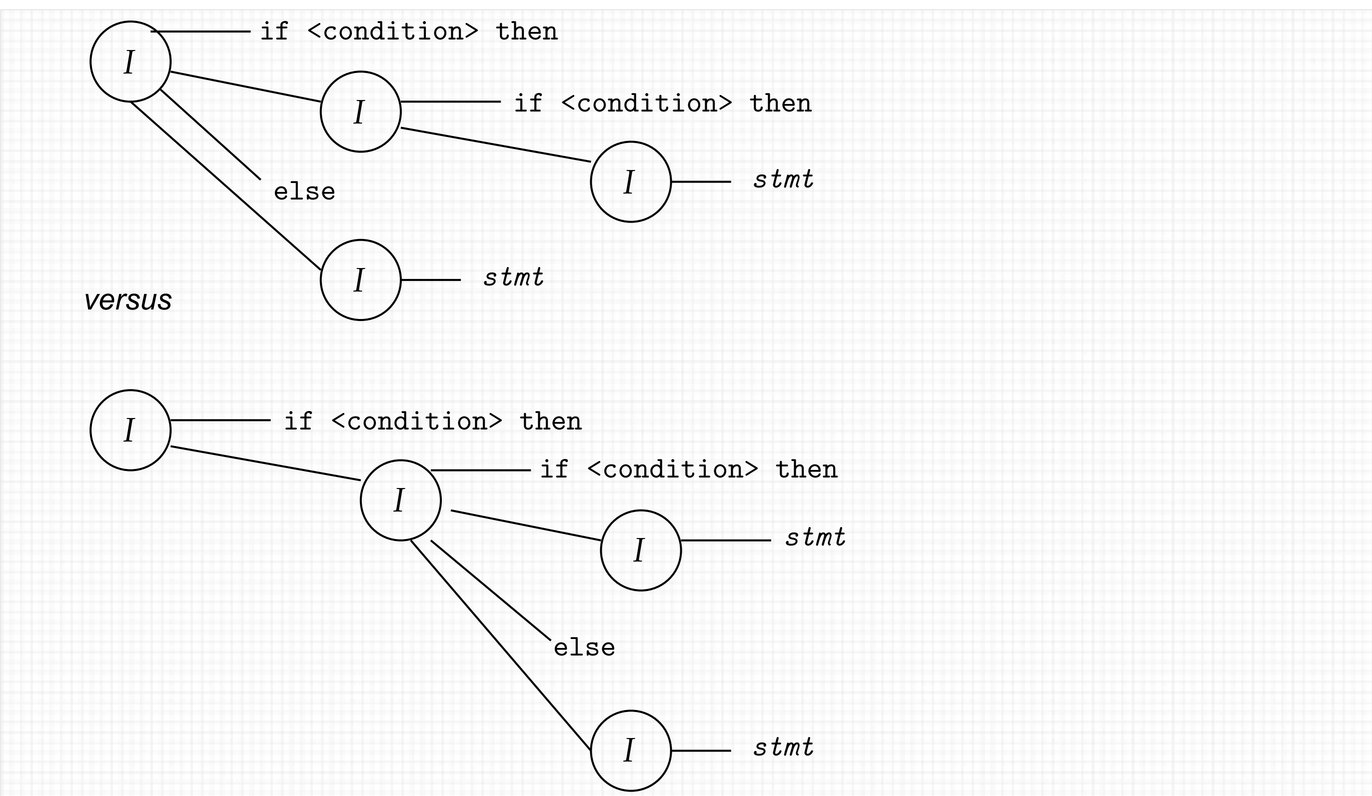
This pops up also in languages that regard statements as their main elements rather than expressions. The rules in C/C++/Java and other languages where an else-branch is optional are typified by these forms, which all allow the ambiguity:

STMT ::= if (EXP) STMT [else STMT], where also STMT ::= { STMT{; STMT} [;] }

STMT ::= if EXP then STMT [else STMT]

`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:



Both trees yield `if <condition> then if <condition> then` *`stmt`* `else` *`stmt`* (where *stmt* is italicized to mean you could put any statement there, or think of it as the STMT nonterminal).

Note that in C/C++/Java/etc., a statement can be a **block**. That is, these grammars have rules like

STMT ::= **{** { STMT; } **}**

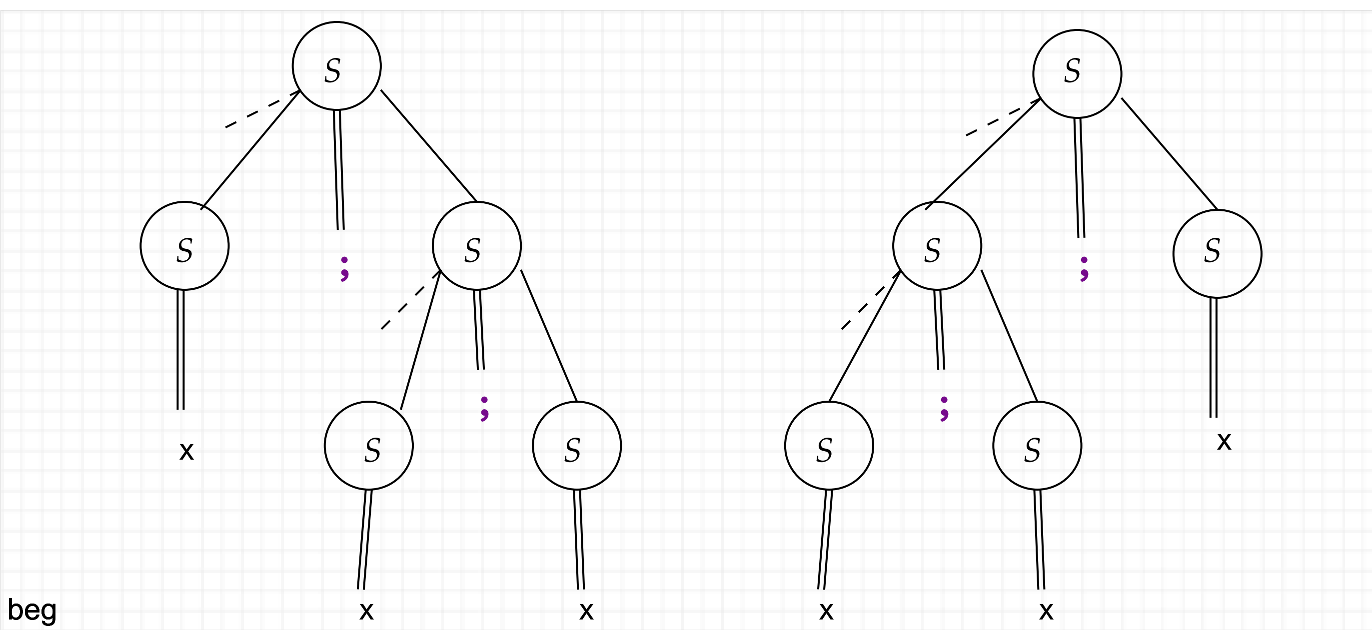
Whoa---the outer braces are the literal ones to define the block; the inner ones are the EBNF metachars for "zero or more". A block *can* be empty in C/C++/Java/etc. Some other languages allow sequences of statements without the braces---that is, lists of statements. There are two ways to implement lists in regular BNF. One is unambiguous, the other ambiguous.

- With a separate syntactic category for lists:
  `STMT_LIST ::= STMT | STMT ; STMT_LIST`
  `STMT ::=` *other-kinds-of-statements...*
- Without: `STMT ::= STMT ; STMT  |` *other-kinds-of-statements...*

The latter is ambiguous for the same reason that EXP ::= EXP <binop> EXP is ambiguous:

Proposition: Any grammar that derives terminal strings via the rule $S\ ::=\ S;S$ is ambiguous.

Proof: Assuming S can derive at least one terminal string x, we get two parse trees for $x;x;x$:



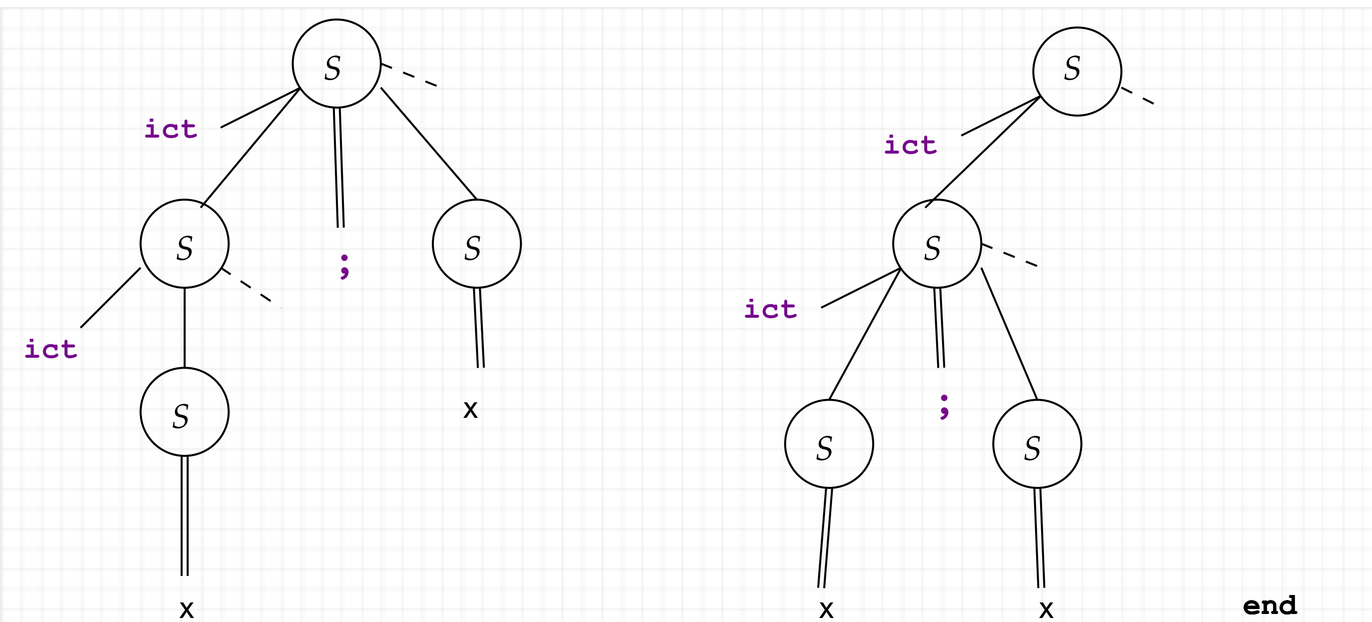
Well *duh*, this just abstracts the expression case we've already seen. The ambiguity holds even if we put a major variable rather than something like `<binop>` in place of the ; part. It can, however, be fixed if we put a simple *begin* marker, even if we don't use a balancing *end* marker:

- Suppose the rule is S ::= **beg** S**;** S instead. Then the ambiguity goes away when we add **beg** at the dashed lines in the trees. The left yield becomes **beg x ; beg x ; x** whereas the right tree yields **beg beg x ; x ; x** instead, which is a different string.

The abstraction pays off a little more with both the "dangling else" ambiguity and the easy---but ignored!---way to fix it. It actually starts with the good rule of the expression fix, except I'll make the cosmetic change of terminal keyword **ict** in place of **beg** to suggest "if (condition) then". Where it goes ambiguous is by making the second S part optional:

Proposition: Any grammar that derives terminal strings via the EBNF rule S ::= ict S [$;$ S] is ambiguous. In BNF terms, the rules S ::= ict S | ict S $;$ S are ambiguous. Again this holds for any terminals or nonterminals in place of **ict** and **;** --- such as **else** in place of the semicolon.

Proof: We get two parse trees for **ict ict x ; x** like so:



- To fix this ambiguity, we could disallow ict S being by itself---that is, allow only the rule S ::= ict S $;$ S. When **;** is "`else`", that is like making the `else` branch of an `if` statement mandatory. Standard ML does this with if-then-else expressions. OCaml and Scala and Python and most other languages do not require `else`.
- But there is another way. We could require that *both* kinds of **if** statement have a closing keyword, such as **fi** or **endif** or just **end**. Here's how **end** fixes it: The left-hand tree yields **ict ict x end ; x end**. The right-hand tree gives **ict ict x ; x end end**. Once again those are different strings.

Many older languages required a closing keyword, but Python and Scala and OCaml do not, as well as

C/C++/Java and Javascript. Why not? There is a universal rule for resolving this ambiguity:

> In **if** `<condition>` **then** **if** `<condition>` **then** `STMT` **else** `STMT`, the **else** branch *always* goes with the *latter*, *innermost* **if**.

A vital reason for this choice will come out when we see more of this ambiguity in OCaml.

More Ambiguity in OCaml

Let's hunt for ambiguity in the OCaml grammar again:

```
EXP  ::=  <value-path>  |  <constant>  |  (EXP)  |  begin EXP end
      |   EXP { , EXP }+.
      |   <prefix-symbol> EXP  |  -EXP  |  -. EXP
      |   EXP <infix-op> EXP
      |   if EXP then EXP [else EXP]
      |   while EXP do EXP done
      |   for <value-name> = EXP (to | downto) EXP do EXP done
      |   EXP ; EXP
      |?  let [rec] VPAT = EXP { and VPAT = EXP } in EXP
```

The rules with red bars have ambiguity. The abstract form made this super-obvious with the rule `EXP ::= EXP;EXP` (not to mention the comma form on line 2). This could be fixed with an extra `EXP_LIST` nonterminal like we did with `STMT_LIST` above. That's what compilers do under the hood, but for human readers, the OCaml people don't care.

The for-loops do not have ambiguity because the leading `while` or `for` keyword and the mandatory closing `done` keyword work like `beg` and `end` in the above examples.

But OCaml uses neither of the above policies for disambiguating the rule for `if` expressions.

OCaml gives the impression of ambiguity in `let` expressions. Let's do the simple form without `rec` where `VPAT` is just an identifier for a variable and there is only one such **binding**:

```
let <ident> = EXP  in  EXP
```

The fact that we can write things like "`let x = 3`" standing alone makes it seem like the rule makes the **in** part optional:

```
let <ident> = EXP  [in EXP]    (?)
```

This plugs right in to the same abstract form as the "dangling `else`" ambiguity. An ambiguous form (substituting in variable names) would then be `let x = let y = EXP in EXP`

The ambiguity would be whether this is grouped as `let x = (let y = EXP) in EXP` or as the expression `let x = (let y = EXP in EXP)`. The same "goes with inner" resolution of the dangling-else ambiguity would dictate the latter. Indeed, if you write

```
let x = let y = 3 in y+1;;
```

it is legal, and you get the same result as `let x = (let y = 3 in y+1);;` That helps you understand why OCaml finally says `val x:int = 4` back to you. But if you try either of

```
let x = (let y = 3) in y+1;;
```

or

```
let x = (let y = 3) in x+1;;
```

you get a `Syntax Error` in both, even though you think `x=4` should be the outcome of both. The reason is that the **`let`** form without **`in`** is not a rule of EXP in the grammar but rather a rule of a different syntactic category, a *definition*. In OCaml this is classed as a primitive case of a **module**, which in turn is a basic **compilation unit**, which can also give an optional `[;;]`. The rule at https://v2.ocaml.org/manual/modules.html#start-section is

```
DEF  ::=  let [rec] VPAT = EXP { and VPAT = EXP }
       |  (other stuff)
```

Well, that's just like the rule for a **`let`** expression without the **`in`** part. But the point us that as a definition, you can't throw something like "`let y = 3`" into the rule for EXP like that. But even if you could, if you allowed (`let y = 3`) grouped like that, you would have a problem of prematurely cutting off the **scope** (text chapter 5) of the variable `y` before you got to the body `y+1`.

Well, the `DEF` rule is like Python and Scala if you used **`def`** rather than **`let`** as the opening keyword. But it is considered to be on a par with a different rule:

`let` `<ident>` **`= fun`** *`parameter_1`* … *`parameter_m`* **`->`** `EXP`

The abbreviation is familiar from recitation examples:

`let` `<ident>` *`parameter_1`* … *`parameter_m`* **`=`** `EXP`

So instead of `let plus1 x = x + 1` you can write `let plus1 = fun x -> x+1`. The latter is like what Scala allows doing with the `lambda` keyword in place of `fun`.

So this is technically not an ambiguity in the `let` part of the OCaml grammar, because it comes from teh different nonterminal `DEF`. But it looks like it and the behavior works the same way as for "dangling else". As my linking an older definition document of OCaml signifies, the actual compilers work with longer grammars than the public one. (Standard ML requires the **`fun`** keyword in function definitions and makes an **`end`** keyword mandatory in its **`let`** ... **`in`** ... **`end`** expression syntax, so there is less of this kind of confusion.)

Building Up Types (in OCaml)

Recitations covered the **base types** in OCaml, including `int`, `float`, `bool`, `char`, `string`, and `unit`. We can now express how OCaml builds up compound types by giving EBNF rules for the major syntactic category of types.

OCaml (like Standard ML) distinguishes between "expressions" made up from its native types and user-constructed type definitions, calling the former (in my all-caps style) TYPEXP and the latter TYCON. The base types are actually classed as type constructors since they (except `float`) are the bedrock of pattern-matching and type inference, so we have to mention TYCON to get the basis. But we'll skip the other stuff in TYCON for now. Again, this is just an illustrative subset of the actual rules in the (public) grammar at https://v2.ocaml.org/manual/types.html#start-section.

```
 TYCON ::= the basic types  |   list   |   lots of other stuff.

TYPEXP ::= '<ident>  |  _  |  (TYPEXP)   ( 'a is like template <A> in C++/Java )
         |  TYPEXP -> TYPEXP              (function type, associates to the right)
         |  TYPEXP { * TYPEXP }+          (tuple type, no left/right handedness)
         |  [TYPEXP] TYCON                (simple example: int list)
         |  (TYPEXP {, TYPEXP}) TYCON
         |  TYPEXP as <ident>             (like typedef TYPEXP <ident> in C/C++)
```

The underscore _ is a "wildcard" type expression and is used like with matching in Scala. The rule `TYPEXP ::= TYPEXP -> TYPEXP` is ambiguous, but we followed the "main takeaway" by stating an association rule for it. The order of giving the rules expresses precedence. Thus, for instance, in

```
int * int -> float
```

the * "binds tighter" than the arrow. That is to say, this is grouped as `(int * int) -> float`, which is a function of a tuple of two integers giving a float result, rather than being grouped as `int * (int -> float)`, which is a tuple like `(3, fun x -> (float_of_int x)/. 2.0)`. "Under the hood" here is an "ETFP"-like grammar that associates function composition right-to-left as done with powering---but groups it loosest rather than tightest.

The beautiful point here is that not only do these inductive grammar rules generate the syntax by which you can write type annotations (after a colon :) if need be, they define how OCaml builds up its entire type system internally to begin with. And when we get to the rest of TYCON, it places that power into user hands, giving the user programming syntax that is like BNF grammar itself. But before creating "*matchable structure*" via TYCON, let's see how we define the patterns usable in matching.

Patterns in OCaml

Let's dive right into the grammar rules before showing how examples conform to them. Again we give a subset of the rules at https://v2.ocaml.org/manual/patterns.html#start-section:

```
PAT ::=  <vname>  |  _  |  <constant>  |  PAT as <vname>  |  (PAT [: TYPEXP])
      |  PAT | PAT
      |  <cname> PAT
      |  PAT {, PAT }+                   (tuple pattern)
      |  [ PAT {; PAT}[;] ]              (pattern for fixed-size list)
      |  PAT :: PAT                      (pattern to handle general-size list)
      |  [| PAT {; PAT}[;] |]            (pattern for fixed-size "value array")
      |  <char> .. <char>
      |  lazy PAT
      |  exception PAT
```

The first line of options are like those we had with ordinary expressions: constant and variable (lowercase) are options, but now also _ for wildcard. The second line says a pattern can have internal BNF-like alternatives and they are the outermost/loosest/highest "operators" in any pattern.

In the third line, <cname> mostly means a ***capitalized*** identifier name. Those come from user-defined type constructors, which includes classes but more primitive stuff first (next week).

The next four lines are the bread-and-butter tuple and list patterns, plus one for arrays. One technical note: The empty list `[]` is classed as a `<constant>`. So is the empty array `[||]`. Note that if we had written the rule for list patterns as `[ {PAT ;} ]` it would have suggested that you could put space between the brackets. *(Voiceover: you can...)* It would also require a final `;` in a nonempty list pattern.

Next we can also allow a range of literal characters as a pattern. The last two lines are just for forward reference, FYI for now. And incidentally, here is the rule for `VPAT`:

```
VPAT  ::=  PAT
   |  <vname> { PARAM } [: TYPEXP] [:> TYPEXP]
   |  <vname> : POLYTYPEXP
```

where we will see the `:>` **type coercion** (which is like `extends` or `implements` in OOP languages) and **polymorphic type expressions** later. (The actual OCaml grammar writes these rules with the "`= EXP`" part of "`VPAT = EXP`" down here, calling the whole thing a *let-binding*.)

Pattern Matching in OCaml

To show how patterns are used, we need only mention two more lines of the rules for expressions:

```
EXP ::=  match EXP with PATMATCH
       | EXP { ARG }+
```

```
PATMATCH ::=  [ | ] PAT [when EXP] -> EXP { | PAT [when EXP] -> EXP }
```

I could have put the line with `ARG` earlier; `ARG` goes right back to `EXP` but with **label** options too. If we ignore the optional `when` feature, and ignore that OCaml doesn't care if you put an unnecessary bar `|` before the first pattern in your `match` body, we can **condense** this into one simplified rule---also showing possible indentation:

```
EXP ::=  match EXP with
            PAT -> EXP
        { | PAT -> EXP }
```

This says that you do need bars to separate multiple patterns used in your match. Let's derive a whole example function that does pattern matching. The example is

```
let rec sumList ell = match ell with
     [] -> 0
   | x :: rest -> x + sumList rest;;
```

As noted before, the syntactic category for this is `DEF` (which comes as a simple case of `COMPUNIT`, which is the start symbol for the whole programming language grammar).

```
DEF ⟹ let rec VPAT = EXP
⟹ let rec <vname> PARAM = EXP            (taking one param from { PARAM })
⟹ let rec sumList PARAM = EXP
⟹ let rec sumList ell = EXP
⟹ let rec sumList ell = match EXP with
         PAT -> EXP
```

```
      | PAT -> EXP                   (taking one extra pattern from { | PAT -> EXP })
⟹ let rec sumList ell = match ell with  (via EXP ⟹ <value-path> ⟹ <vname> ⟹ ell)
        PAT -> EXP
      | PAT -> EXP
⟹ let rec sumList ell = match ell with
        <constant> -> EXP
      | PAT -> EXP
⟹ let rec sumList ell = match ell with
        [] -> EXP
      | PAT -> EXP
⟹ let rec sumList ell = match ell with
        [] -> 0
      | PAT -> EXP
⟹ let rec sumList ell = match ell with
        [] -> 0
      | PAT :: PAT -> EXP
⟹^2 let rec sumList ell = match ell with
        [] -> 0
      | <vname> :: <vname> -> EXP
⟹^2 let rec sumList ell = match ell with
        [] -> 0
      | x :: rest -> EXP
⟹ let rec sumList ell = match ell with
        [] -> 0
      | x :: rest -> EXP + EXP
⟹ let rec sumList ell = match ell with
        [] -> 0
      | x :: rest -> EXP + EXP ARG
⟹* let rec sumList ell = match ell with
        [] -> 0
      | x :: rest -> x + sumList rest
```

More lines for expressions---point is that pattern matching is basic to defining functions:

```
EXP ::=  match EXP with PATMATCH
      |  function PATMATCH
      |  fun { PARAM }+ [: TYPEXP] -> EXP
      |  try EXP with PATMATCH
```