CSE305: Programming Languages --- Week 2 Lectures

[Tuesday's lecture first finished the Week 1 notes then reached here in its second half.]

A little more history leading into the subject of **Grammars**

- Early/mid 1800s: Charles Babbage and Ada Lovelace conceive Universal Computation from a machine standpoint, mostly numerical.
- Late 1800s: mathematicians started distinguishing concrete/constructive/calculational mathematics from general mathematics.
- Early 1900s: formalization of recursive processes that are symbolic and **generative**. This employed abstract rewriting rules of strings---an idea far older.
- Bertrand Russell and Alfred Whitehead, Principia Mathematica (PM): a system by which to compute proofs. Developed ("**Ramified**") **Type Theory** to resolve Russell's Paradox. PM is a calculus for generating proofs of theorems...
- ...which Kurt Gödel showed in 1931 cannot capture all of mathematical truths---not in any one effectively axiomatized system of logic.
- Early 1930s: Alonzo Church developed the (untyped) Lambda Calculus. This is really the first programming language---it underlies (basic) **Lisp** (1950s). Soon came the Typed Lambda Calculus, and later came strongly typed versions of Lisp such as **Scheme**.
- (Church was awarded an honorary doctorate by UB in 1990.)
- 1936: Alan Turing conceives the **Turing Machine** and proves it equivalent in computing power to the lambda calculus and various other generative systems. Comes to Princeton to be supervised by Church, obtaining his doctorate in 1938---just in time for WW II.
- 1940s: computer machine language developed.
- 1950s: successively higher-level languages conceived and designed: **FORTRAN**, **COBOL**, **Algol** (into the 1960s).
- 1950s: Noam Chomsky (still alive!) uses a simplified model of generative processes called **Context-Free Grammars** to model *human* language. 1957 book *Syntactic Structures*.
- Early 1960s: Shortcomings of Chomsky's **CFG**s as a model of human languages emerge...
- ...but they turned out to be *exactly the right backbone* for programming languages.
- As systemized by John Backus and Peter Naur for Algol-60, **BNF** does not only define the basic syntax, it guides the compilation process. (It does not do type-checking by itself---but the compilation stage of doing so comes right on top of the BNF **parsing** stage.)
- The ML family of languages---including OCaml---not only have a rigorous type-checking system, they have a **full semantics** for defining the intent of programs and making sure execution matches it. (Example.) We will not go that far: to paraphrase a line from the musical *Hamilton*, 'syntax is easy; semantics is harder.' So we start with syntax.

Grammars (specifically, Context-Free Grammars)

Chomsky's Q: How is it that we can speak and understand sentences we've never heard before?
Our Q: How can a compiler understand and translate arbitrarily-big programs that have never been written before?

Secret: *by generating them via inductive definitions*, and the reverse process, which is **parsing**.

Indeed, what is IMHO remarkable is the lack of rules for units higher than a sentence. We have the notion of "paragraph" but it is highly flexible. Newspapers keep them short, blogs try to, but some famous novels run paragraphs for pages and pages. You may have been taught that an essay is composed of an introduction, body, and conclusion, and there are prescribed formats of kinds of business letters, for instance. But if you violated those higher-level rules, it wouldn't make what you wrote unintelligible. Your boss would still get it.

Whereas, meant out be really what to may impossible it structure readers you Yoda
text you or and violate of even order phrase rules if word the for figure .

Whereas, if you violate the rules of word order and phrase structure it may be
impossible for readers or even Yoda to figure out what you really meant .

What struck Noam Chomsky in the 1950s was that although different human languages have different rules for sentences, the natures of those rules are much the same. To a (debatably) large extent, they can be given as CFG rules. One result was an effort toward systemabstrimplification of how grammar was taught in schools. When I was in primary school, I recall a book that had

$$S \rightarrow N\ V$$

The intent, rendered more accurately in BNF style as in the text, was

```
<sentence>  ::=  <noun-phrase> <verb-phrase>
```

That rule applies to the great majority of sentences in English--where `<verb-phrase>` can expand to allow direct and indirect objects and other forms that can involve more noun phrases. Does every full sentence follow that rule? At least every non-interrogative sentence? Think about it!

In English we can further expand:

```
<noun-phrase> ::=  <noun>  |  <article> <noun>
                           |  <adjective> <noun-phrase>
                           |  <noun-phrase> <prep-phrase>
```

The rule `<noun-phrase> ::= <adjective> <noun-phrase>` allows you to put one or any number of adjectives before a noun---with the zero option coming in if you use one of the first two rules immediately. In "extended BNF" notation you can use square brackets to indicate optional stuff and braces for zero-or-more (just like Kleene star), so we could write more compactly:

```
<noun-phrase>  ::=  [<article>] {<adjective>} <noun> {<prep-phrase>}
```

(Actually, this is not equivalent to the above grammar---it fixes an error in the placement of articles that actually requires having a separate variable saying the article is optional with the rule `<art-opt-noun-phrase> ::= <article> <noun-phrase>  |  <noun-phrase>`.)

An example taking the optional article, the zero option for adjectives, and one prepositional phrase is "the cat in the hat". We could extend it to be "the cat in the hat with a bat." It is curious that those phrases are modifiers like adjectives are but come after the noun. We could have said, "the hat-wearing, bat-carrying cat."

Let's just use N for noun, N_P for noun-phrase, and A for adjective. The rule

$$N_P \rightarrow A\, N_P \mid N$$

places no limit on the number of adjectives we can have before a noun. It might seem sensible to have a limit like 3 or 4, but it is actually both *simpler* not to impose a limit, and more indicative of how we talk---or can talk---especially in the heat of the moment. For instance,

"You are a dirty rotten stinking lying skunk!"

This applies $N_P \rightarrow AN_P$ four times before terminating with $N_P \rightarrow N$ at the word "skunk". Now French has a different rule, basically $N_P \rightarrow N_P\, A$ so that adjectives come after the noun (but not exclusively, as we'll see). Let's try insulting "Pepé Le Pew" by translating this to French on Google...

[Try the above on Google Translate. You may get some surprises, such as GT thinking that "lying" means lying down. Change "lying" to "fibbing" or "untruthful". Then try translating the French back to English (but if you get the word "putain" in the French, don't). See if you can get something that keeps coming back the same when you go back and forth, so that GT's French and English agree on what is being said.]

What English and French share is not the vocabulary or rules but the sameness of the nature of the rules. That sameness extends to non-Indo-European languages. Isolated language communities were found to have rules that can be modeled to a similarly large extent by CFG rules. The CFG rules don't catch everything, but they catch a lot, and they appear to matter to our brains in a way that ***precedes*** the meaning of the words. Chomsky's famous sentence to illustrate this is:

Colorless green ideas sleep furiously.

It makes poetic sense, despite the first two words contradicting each other, and the last two words... Whereas, there are times when even if we completely know in advance what a speaker is going to say we can still get uptight if we have to wait...

Key Example: Expressions.

Expressions are a major part of every computer language. Here is an inductive definition that does not pre-suppose any one lexical notation for expressions.

Base: Any constant or variable is an expression.

Ind1: If E is an expression, then applying the unary '–' operator to E also gives an expression.

Ind2: If E1 and E2 are expressions, then applying any one of the binary operators `+,–,*,/` to E1 and E2 also gives an expression.

Example: To generate E = `b*b – 4*a*c`, apply the rules as follows:

E = E1 – E2 (by **Ind2**), where

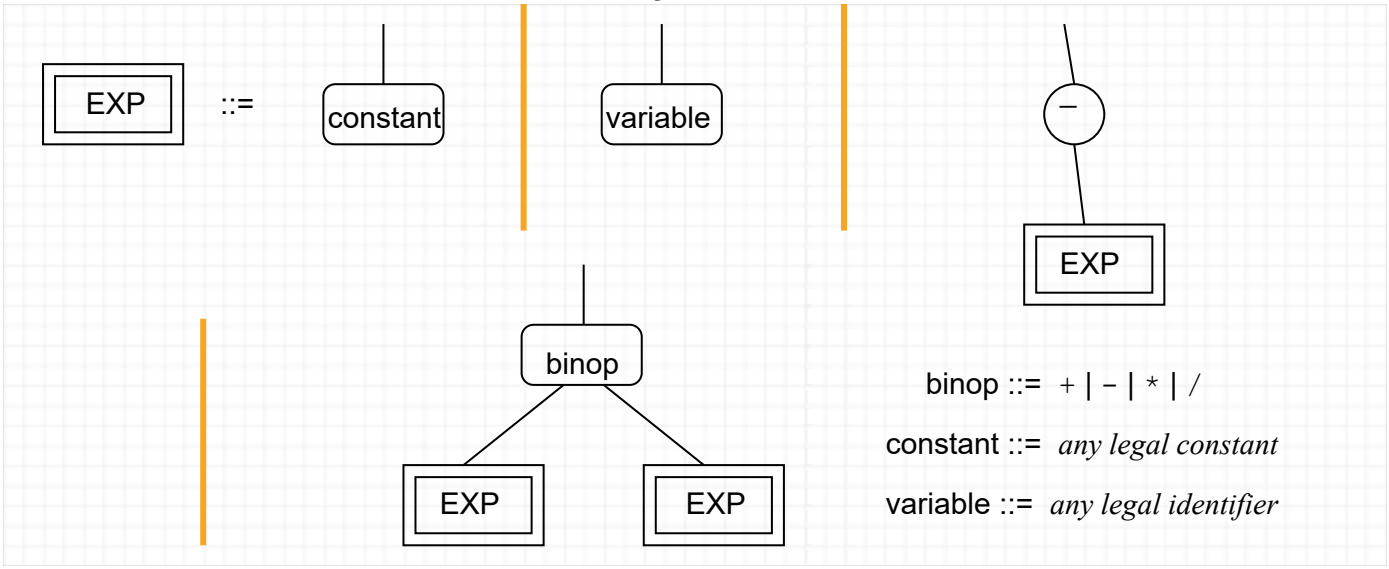
E1 = `b * b` (by **Ind2** on two base exprs "`b`"), and

E2 = E3 `* c` (by **Ind2**), where

E3 = `4 * a` (again by **Ind2**---we didn't need to use **Ind1** for this expression).

This is "top-down"; a "bottom-up" derivation would start with E3 = `4 * a` or with E1 = `b * b`.

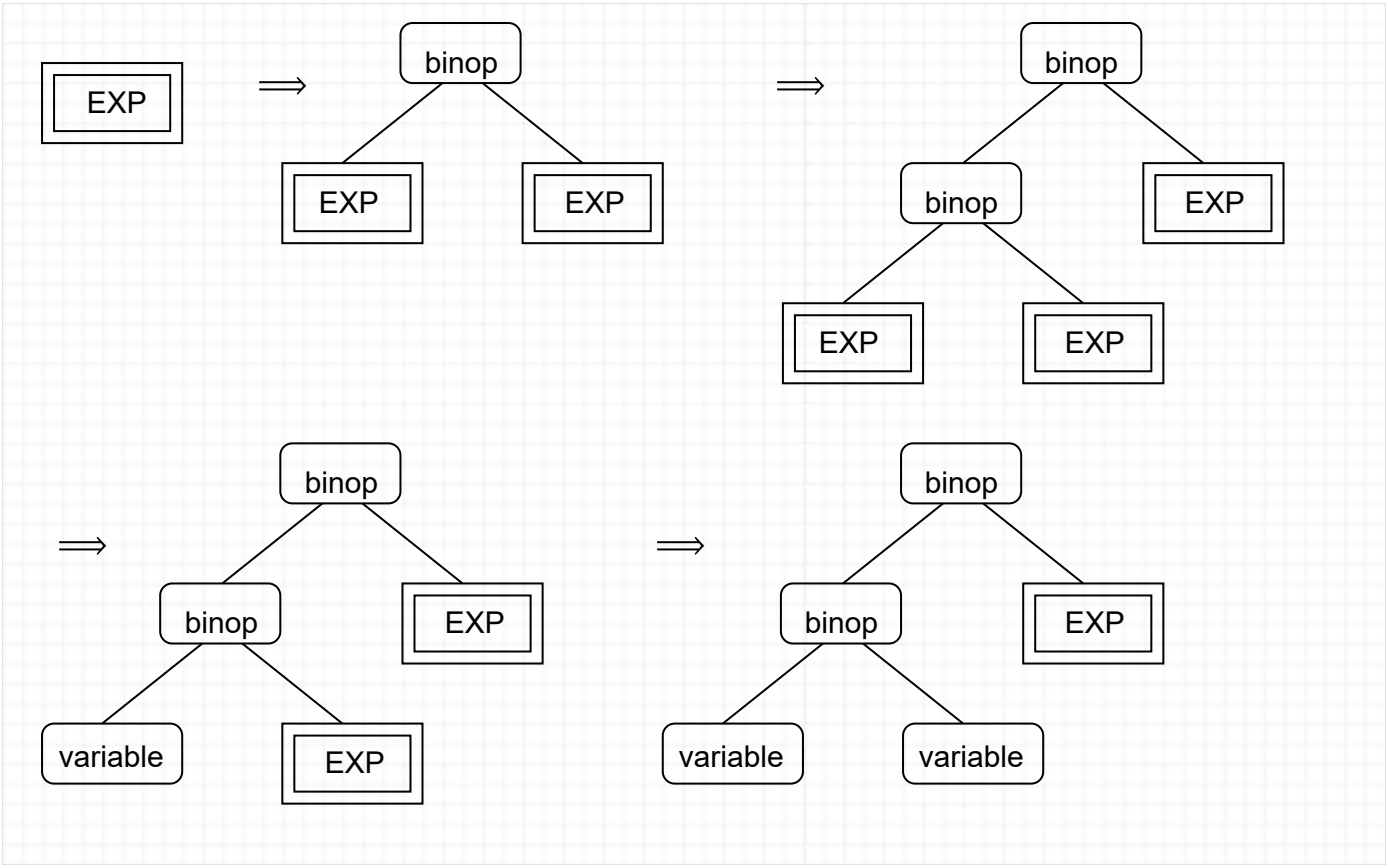
We can re-write the above inductive definition to generate trees:



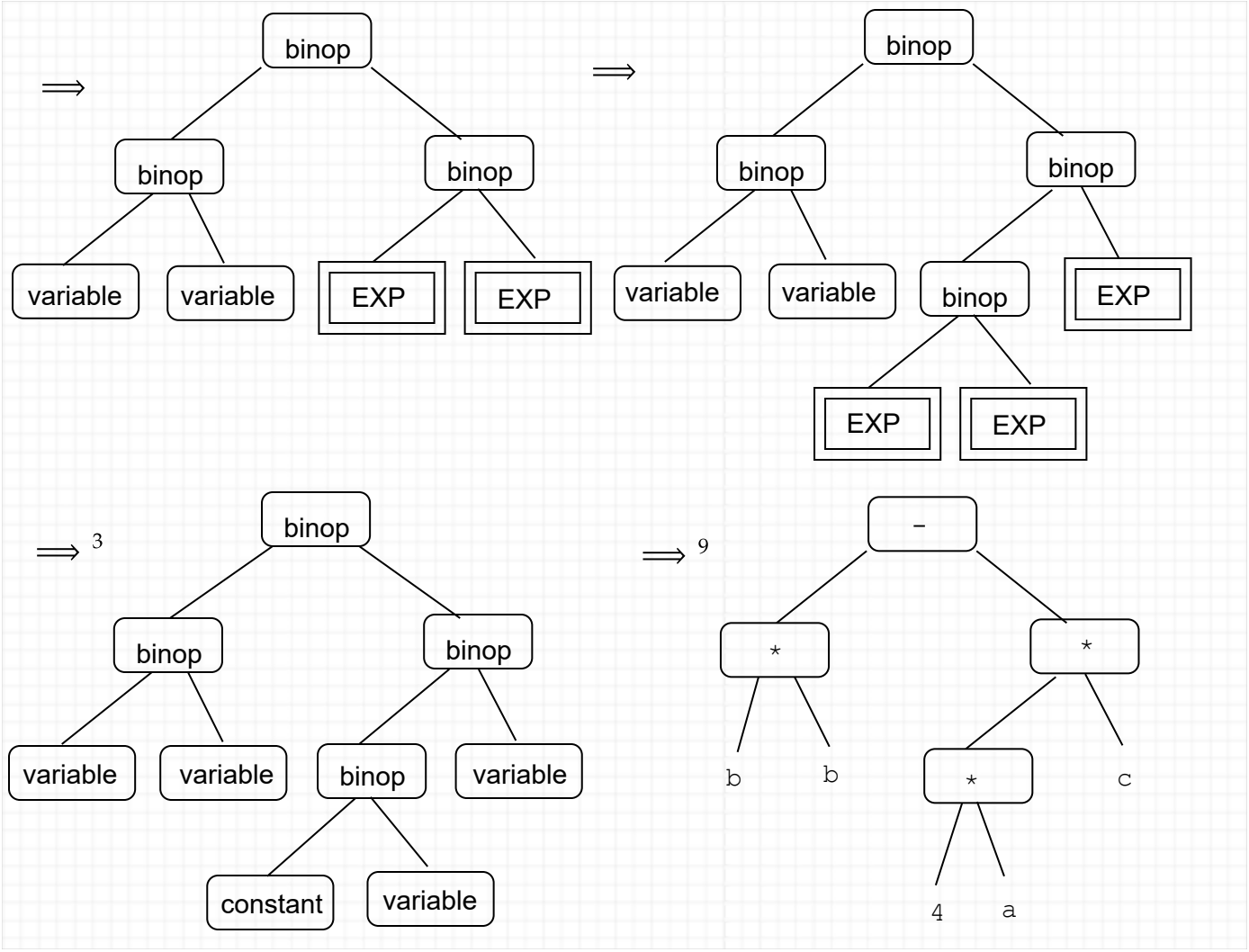
[The Tue. 2/7 lecture ended here; I will pick up with the above figure on Thu. 2/9.]

I could have made a separate **syntactic category** "unop" for unary operator. Then I could have allowed unary + as well as unary – as an option. When a syntactic category is going to be derived directly to some literal(s), whose details we might not care about so much, we regard it as a "minor category" and don't emphasize it so much in the figure. Text: Minorcategories give **lexemes**.

Now we can **derive** `b*b – 4*a*c` in tree form in top-down fashion as follows (when it's clear which node of a tree is the **root**, you don't need to show the stick going into it.):

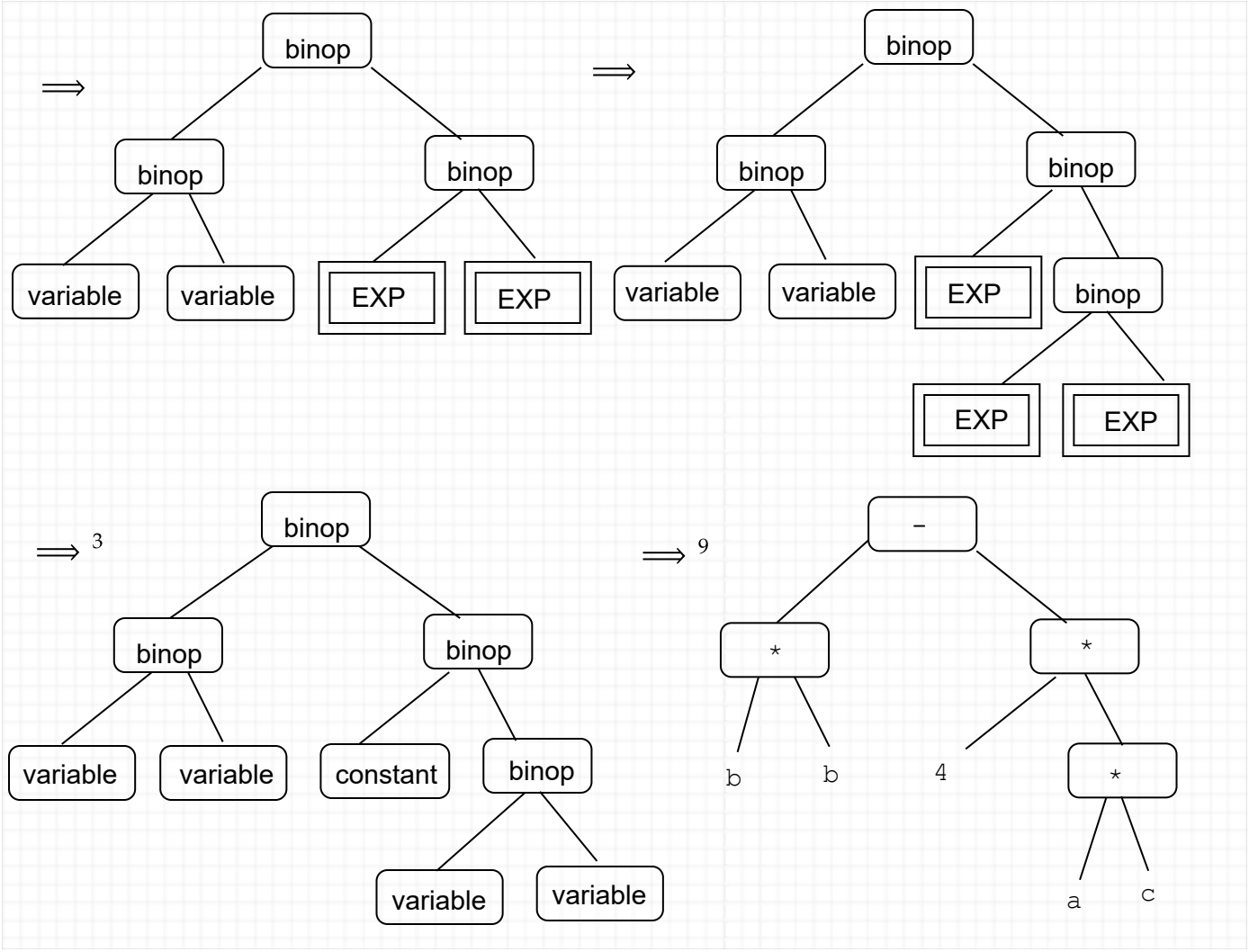

EXP
binop
EXP
EXP
binop
binop
EXP
EXP
EXP
binop
binop
EXP
variable
EXP
binop
binop
EXP
variable
variable

(continued)



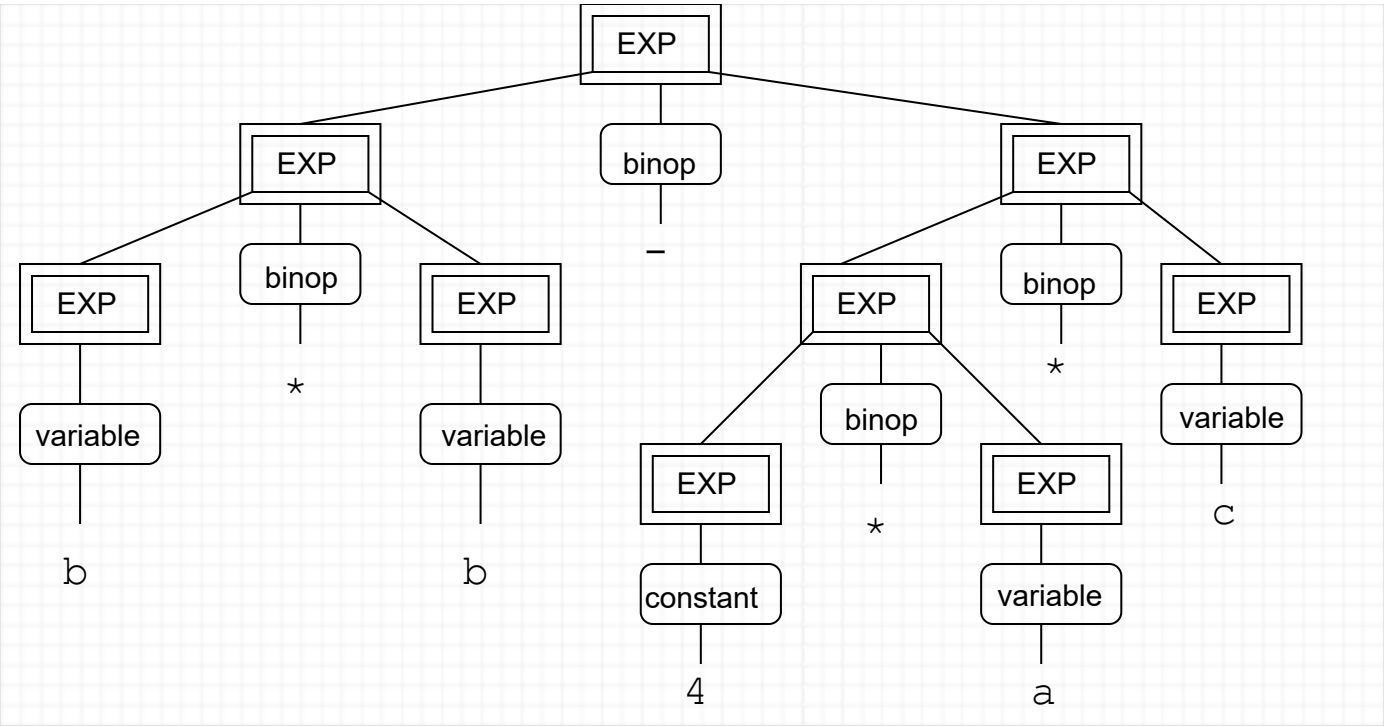
Here we combined three derivation steps to fix the last three sub-expressions to their minor syntactic categories. Then it technically took nine more steps to substitute each minor category by a corresponding literal. It so happened that the first two "variable" nodes went to the same literal, `b`, but that is not mandatory---the other two occurrences of "variable" went to `a` and `c`, respectively. Each node is free to derive whatever is allowed to it, regardless of what other nodes do. This is the "**context-free**" idea. The final product is the **expression tree** for the expression `b*b-4*a*c`.

Note that the expression tree specified how the `4*a*c` part should be grouped. It made `4*a` as a separate sub-expression, in away that expresses the intent for it to be evaluated first. That is, it groups it as if the whole expression were `b*b-(4*a)*c`. If we wanted `b*b-4*(a*c)` instead, we would have needed to apply the "binop" rule to the "EXP" at lower right:



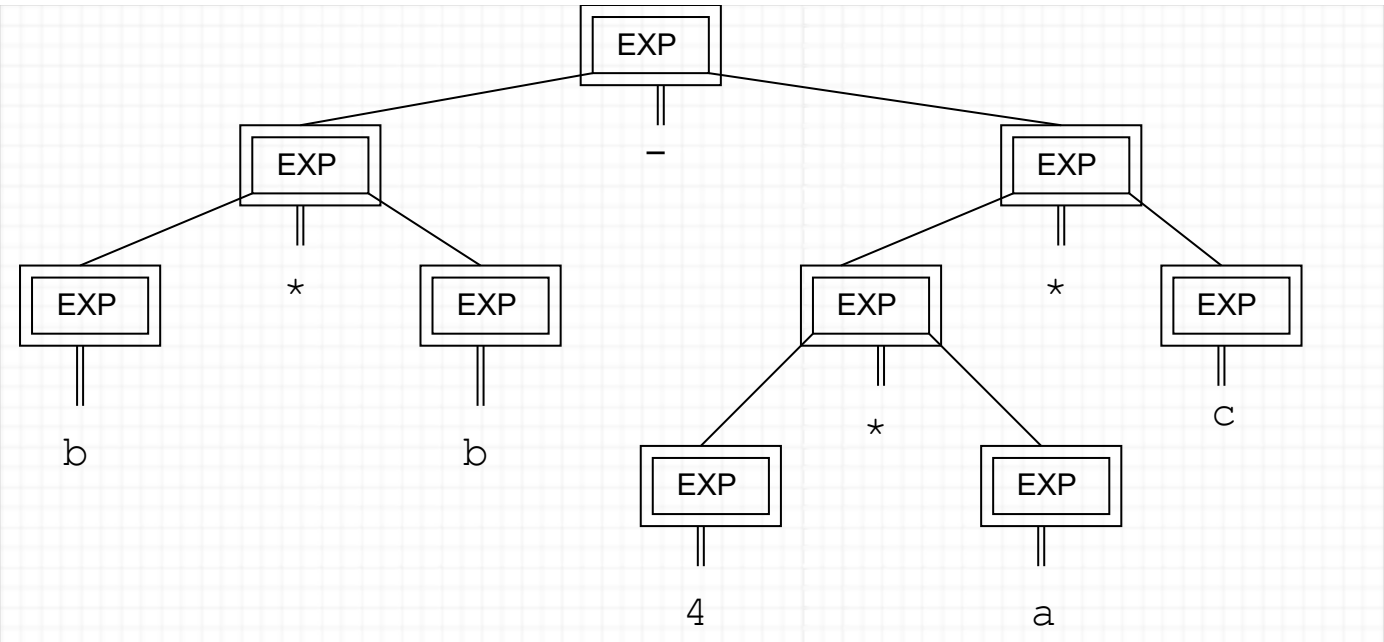
More KWR style notes: I keep circles around **interior nodes** of trees---which in expression trees are always the operators---but skip them around the **leaves** of the trees, which are variables and constants *and maybe other keywords and punctuation*. Sebesta doesn't use any circles, but the angle brackets around the syntactic categories are almost as good.

Not just a style note, however: we can convert the expression tree into a **parse tree** if we keep all the occurrences of EXP that we expanded in the tree:



To avoid confusion with mathematical variables in expressions themselves, and identifiers that are program variables more generally, the syntactic category (text sometimes "syntactic structure" similarly meaning the concept not just the nonterminal) names are often called **nonterminals** rather than "grammar variables." The literals---that is, the characters or strings in the leaves---are called **terminals**.

In "my style", it is OK to cut out the minor nonterminals---and abbreviate any chain of one-for-one substitutions---by a double line. So the parse tree looks a little trimmer this way:



This also makes it clearer that if you roll up the leaf terminals into their parent nodes, you get the original expression tree back again.

CFG and BNF

We can express the basis and inductive rules for building up our expressions in a much more compact symbolic form like so (I've distinguished terminal symbols in orange bold):

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

Here the "`::=`" symbol (is read "can be a", and the "`|`" is read "or a". This is called a **context-free grammar** (CFG) in **Backus-Naur form**, or **BNF grammar** for short. CFG notation tends to use capital letters for nonterminals and arrows → rather than `::=`. *KWR style*: I find it cleaner to use CFG caps for the major syntactic categories and angle brackets for the minor ones. One reason is that you often want to treat a minor item as if it were a terminal, without fussing over how to expand it (so the angle brackets come to mean "not expanded further"). Hence I will tend to write grammars in this hybrid style:

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

Applying these rules creates **derivations** typified by the expression above.

```
EXP ⟹ EXP <binop> EXP
    ⟹ EXP <binop> EXP <binop> EXP
   ⟹ b <binop> EXP <binop> EXP
   ⟹ b * EXP <binop> EXP
   ⟹ b * b - EXP
   ⟹ b * b - EXP <binop> EXP
   ⟹ b * b - EXP <binop> EXP <binop> EXP
   ⟹ b * b - 4 <binop> EXP <binop> EXP
   ⟹ b * b - 4 * EXP <binop> EXP
   ⟹ b * b - 4 * a <binop> EXP
   ⟹ b * b - 4 * a * EXP
   ⟹ b * b - 4 * a * c
```

(Here we availed the shortcut of skipping `<constant> or <variable>`.) Every line in this derivation is a legal **sentential form** from the starting EXP nonterminal. The above is a **leftmost derivation**. We can also do the corresponding **rightmost derivation**.

```
EXP ⟹ EXP <binop> EXP
    ⟹ EXP <binop> EXP <binop> EXP
    ⟹ EXP <binop> EXP <binop> EXP <binop> EXP
    ⟹ EXP <binop> EXP <binop> EXP <binop> c
    ⟹ EXP <binop> EXP <binop> EXP * c
    ⟹ EXP <binop> EXP <binop> a * c
    ⟹ EXP <binop> EXP * a * c
    ⟹ EXP <binop> 4 * a * c
    ⟹ EXP - 4 * a * c
    ⟹ EXP <binop> EXP - 4 * a * c
    ⟹ EXP <binop> b - 4 * a * c
    ⟹^2 b * b - 4 * a * c
```

However, programming language references have gravitated toward using *neither* of these styles. Instead they use *italics* for syntactic categories in a way pioneered by Brian Kernighan and Dennis Ritchie's brilliantly economical book *The C Programming Language*. Here is OCaml using it. Well, what they call "BNF-like" notation includes the square brackets [...] for "optional" and braces {...} for "zero or more", which Sebesta calls "EBNF" for "extended BNF". Other common EBNF styles use a subscript `_opt` for optional elements and/or a superscript star * (called "Kleene star") to mean zero-or-more repetitions of an element in braces (or parens or other brackets), with a superscript + meaning one-or-more. With EBNF, you have to distinguish when brackets or braces or * or + are literal terminal symbols, or when they are the "meta-symbols" of EBNF. A third extension allowed by EBNF is "internal choices" exemplified by (FOO | BAR), where the parentheses and | are "meta" and one of FOO or BAR must be substituted.

The actual OCaml grammar for expressions includes a lot more rules and syntactic categories. Let's take a look. (Magnifying the page is needed to tell some of the brown terminals from the EBNF meta-symbols left in black.) This subset of the rules should look fairly familiar. In place of `<variable>`, OCaml has `<value-path>`, which allows for paths thru modules and is otherwise a `<value-name>`, which in turn enforces that regular variables begin with a *lowercase* letter.

```
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
```

This is almost self-contained: VPAT adds a few extra value-name options to PAT which is a major separate syntactic category of OCaml. We will encounter the aspects of PAT used in pattern matching next week. The pleasant surprise for me is how close this is to basic expression examples. It is also *hugely ambiguous*---we'll cover this next week---and thereby hangs a tall tale: this is not the actual grammar used by any OCaml compiler. It is a "human reference grammar."

Here are some interesting points to note:

- Putting any expression inside literal parentheses makes it an "atomic unit" just like any variable or constant. (We'll see why this matters when covering **precedence** next week.)
- The `begin` ... `end` form is equivalent to parentheses but looks better using indentation with compound expressions that span multiple lines of code.
- That's right: if-then-else "statements" and the "while" and "for" kinds of looping "statements" are *expressions* in OCaml. Their value is the if/else branch taken or the **yield** of the last expression executed as the loop exits. Scala behaves the same way.
- OCaml classes unary minus (in both integer and floating-point forms) apart from other prefix operators. This resolves a potential clash with binary minus. Standard ML avoided such a clash by making unary minus a separate symbol: the tilde ~.
- An expression returning `()`, which is called "unit", is intended to act like a statement. The `()` is classed as both a constant and a basic "type constructor" (alongside `[]` which is the only way to write the empty list---there is no keyword `nil` as in Standard ML or `Nil` as in Scala). In both ways, "unit" is like "`void`" in C/C++/Java, but it is more versatile.
- OCaml has two ways to sequence expressions: comma or semicolon. The `;` is the time-sequential operator and works like the "comma operator" in C/C++: its value is the value of the last expression in the chain. You will get a warning if the previous expressions return anything other than unit.
- The comma works like in Python: it forms tuples. You can enclose the tuple in parentheses by doing, say, EXP $\Longrightarrow$ `(`EXP`)` $\Longrightarrow$ `(`EXP`,` EXP`,` EXP`)` but this is optional. It is curious that the OCaml grammar provides EXP ::= EXP `;` EXP but not EXP ::= EXP `,` EXP as a rule, since the latter could be iterated to have the same effect as EXP ::= EXP { `,` EXP }$^+$. The official rule may intend its use of EBNF braces to signify that an unambiguous BNF list form is intended, whereas the EXP ::= EXP `;` EXP rule is ambiguous from the get-go.
- The EBNF square brackets in `if EXP then EXP [else EXP]` signify that having an "else" branch is optional. This is a difference from Standard ML where the "else" branch is required---as is the "`:`" part of the if-then-else expression `(foo ? bar1 : bar2)` in C/C++/Java.
- The EBNF required-choice notation shows up in `(to | downto)` of the for-loop. It would not be cricket to write this as `[down]to` because that would be splitting up a keyword.

Now let's see a different way of doing expressions---just the numerical part again---that is typical of "reference grammars" for C/C++/Java syntax.

```
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
E4  ::=  P++ | P-- | P
P   ::=  (E) | <constant> | <variable>   (etc.)
<assignment_op> ::= = | += | -=   (etc.)
<binop>  ::=  == | != | + | - | * | /   (etc.)
```

[The Thursday 2/9 lecture ended here; Tuesday 2/14 will pick up here, then go to week 3 notes.]

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
```