Context-Free Grammars

- The second phase of a compiler's front end is ***parsing***.

- The parser takes as input the output of the scanner: a stream of lexemes from the source code, each annotated with its syntactic category. Its job is to determine the syntactic structure of the source program—that is, to determine how the individual words in the source code fit together into phrases in the source language—and to build an intermediate representation (IR) of the source program for use in later phases of the compiler.

- Part of the work done by the parser is determining whether the input source code is even a valid program. The user must be alerted if it isn't.

- For the parser to be able to accomplish its goals, it needs a specification of the syntax of the source language. The most commonly used model for such specifications is the ***context-free grammar*** (or ***CFG***).

- Indeed, CFG-based automated parser generators are widely used, probably even more so than automated scanner generators.

- For this reason, CFGs are an important tool for the compiler writer, so we will review them here.

Why not use regular expressions?

- You might be asking why we need an additional specification for a language's syntax, when we've already seen that regular expressions (REs) work well for recognizing the language's microsyntax.

- To see why REs won't work for specifying language syntax, consider the problem of recognizing one simple kind of construct: algebraic expressions over variables using the operators `+`, `-`, `*`, and `/` and allowing the use of matched parentheses for grouping expressions and overriding operator precedence, e.g. `(a+b)*c`.

- Let's try to write a regular expression to recognize such phrases. Assuming we have a Flex-like name `ID` defined to use as shorthand for the RE for recognizing valid identifiers, we can start by defining an RE (in Flex notation) to recognize expressions *without* parentheses:

  ```
  {ID}([+-/\*]{ID})*
  ```

- Now, let's try to add in parentheses. In a correctly-parenthesized expression, any identifier can have an opening parenthesis before it or a closing parenthesis after it. This leads us to an RE like this, which looks like the one above with added parentheses (in bold):

  \(?{ID}([+-/*]**\(?**{ID}**\)?**)*

- This RE matches all valid parenthesized expressions, e.g. `(a+b)*c`, `a*(b+c)`, etc. However, it also matches lots of *invalid* parenthesized expressions, such as `(a+b*c` and `a+b)*c`.

- The problem here is that regular expressions cannot be written to recognize arbitrarily-nested paired constructs, like parentheses (or braces, brackets, `if-else`, etc.). Indeed, as our textbook states: DFAs cannot count.

Context-free grammars

- Context-free grammars (CFGs) are able to recognize more complicated constructs than REs, such as nested, paired parentheses.

- A CFG is a set of rules that describes how to form valid sentences (i.e. valid programs) in a particular language. The rules in CFGs are called ***productions***. Each production describes how a certain language construct can be derived from other language constructs.

- Below is a very simple CFG that contains two productions:

  1 *EatingSound* → `nom` *EatingSound*
  2 | `nom`

- The first production can be interpreted as saying *EatingSound* can consist of the word `nom` followed by more *EatingSound*.
    - Formally, the → symbol is read as "derives".

- The second production can be interpreted as saying *EatingSound* can consist of the only word `nom`.

- In these productions, *EatingSound* is called a ***nonterminal symbol***. Nonterminal symbols serve as syntactic variables and serve as placeholders for actual words in the language (like `nom`), which are called ***terminal symbols***.
    - When using a CFG to describe a real programming language, terminal symbols correspond to the syntactic categories comprising the language's microsyntax.

- Together, the productions in a CFG describe how sentences in the language can be generated, as follows:
    - One specific nonterminal is designated as a goal/start symbol representing all possible sentences in the language. We start with a prototype sentence containing only this goal symbol.
    - A nonterminal α in the current prototype sentence and a production $\alpha \rightarrow \beta$ are chosen, and α is rewritten with β in the prototype sentence.
    - This rewriting is repeated until the prototype sentence contains no more nonterminals, only terminals. At this point, the prototype sentence is a valid sentence in the language.

- The prototype sentence can contain a mixture of terminal and nonterminal symbols. We call this sentence a ***sentential form*** if it occurs in some step of a valid derivation computed as described above.

- For example, in our simple CFG above, *EatingSound* must be the goal symbol, since it is the only nonterminal. Starting with a sentential form containing only this goal symbol, we can apply the grammar's second production to generate the sentence `nom` via the following derivation:

Rule	Sentential Form
start	*EatingSound*
2	`nom`

- Alternatively, we could derive the sentence `nom nom nom` via the following derivation:

Rule	Sentential Form
start	*EatingSound*
1	`nom` *EatingSound*
1	`nom nom` *EatingSound*
2	`nom nom nom`

Revisiting algebraic expressions

- Now that we know a little bit about CFGs, let's revisit the algebraic expression problem from above that demonstrated the shortcomings of REs.

- Indeed, it's quite simple to define a CFG to capture the algebraic expression language:

1	*Expr*	→	`(` *Expr* `)`
2		|	*Expr Op* `name`
3		|	`name`
4	*Op*	→	`+`
5		|	`-`
6		|	`*`
7		|	`/`

- It should be clear that this CFG can never derive an expression with unbalanced parentheses: the only production in which parentheses are introduced is 1, and this production generates exactly one pair of matched parentheses.

- The grammar above does have shortcomings. In particular, the language it represents does not contain expressions that *end* with a parenthesized term, like `a*(b+c)`, unless the entire expression is parenthesized. By omitting such expressions, the grammar is unambiguous.
    - We will see in a bit what it means for a grammar to be unambiguous.

- To represent such expressions in the language defined by this grammar, we can simply reorder the terms, e.g. to `(b+c)*a`. We will see later how to write an unambiguous grammar that captures all algebraic expressions.

Standard derivation strategies and parse trees

- A valid sentence in the language specified by a CFG may be possible to derive via many different production sequences.

- However, as we move towards using CFGs to power parsers, it will be useful to have a systematic way to discover a sentence's derivation (or to discover that it doesn't have a valid derivation) so the process of doing so can be automated.
- There are two different commonly-used systematic derivation strategies parsers typically employ:
    - A ***leftmost derivation*** rewrites the leftmost nonterminal in the sentential form at every step.
    - A ***rightmost derivation*** rewrites the rightmost nonterminal in the sentential form at every step.

- For example, using the CFG from above for algebraic expressions, the leftmost derivation of `(a+b)*c` is as follows:

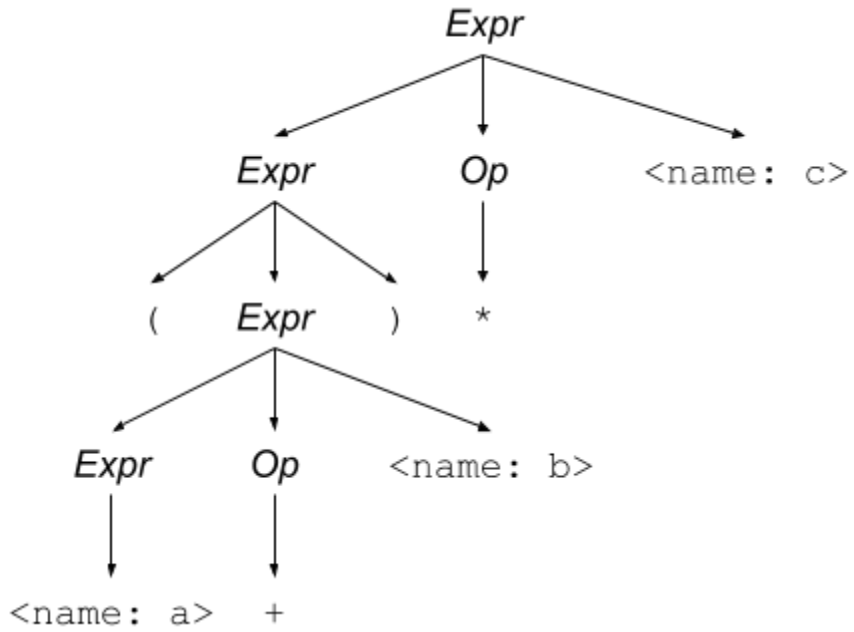
Rule	Sentential Form
start	*Expr*
2	*Expr Op* `name`
1	`(` *Expr* `)` *Op* `name`
2	`(` *Expr Op* `name` `)` *Op* `name`
3	`(` `name` *Op* `name` `)` *Op* `name`
4	`(` `name + name` `)` *Op* `name`
6	`(` `name + name` `)` `* name`

    - Remember, CFGs deal with syntactic categories, not with lexemes, which is why the identifiers `a`, `b`, and `c` don't appear in the derivation here.

- The rightmost derivation of `(a+b)*c` uses a different sequence of productions to achieve the same result:

Rule	Sentential Form
start	*Expr*
2	*Expr Op* `name`
6	*Expr* `* name`
1	`(` *Expr* `) * name`
2	`(` *Expr Op* `name ) * name`
4	`(` *Expr* `+ name ) * name`
3	`( name + name ) * name`

- A ***parse tree*** can be drawn to represent the rules applied in a derivation. Somewhat comfortingly, the parse tree for both of the derivations of `(a+b)*c` is the same:



- Indeed, any valid derivation of `(a+b)*c` will have the same parse tree because our CFG for expressions is ***unambiguous***. Indeed, to say that a grammar is unambiguous *implies* that it has only one parse tree. An unambiguous grammar also has only one leftmost (or rightmost) derivation.

- An ***ambiguous*** grammar, on the other hand, has more than one parse tree and more than one leftmost (or rightmost) derivation.

- When a language has an ambiguous grammar, a program in that language can have multiple meanings. A compiler can't generate code for such a program, since it can't choose between multiple meanings. Thus, it is important that the grammars we define for our languages be unambiguous.

Example of an ambiguous grammar: if-else

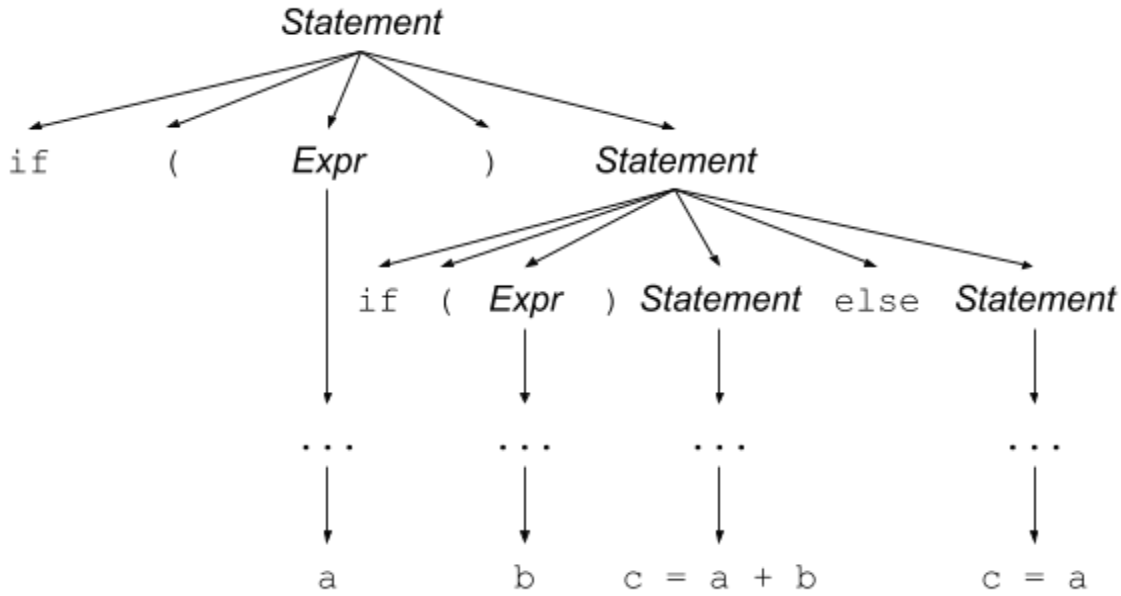
- As an example of an ambiguous grammar, consider a CFG for parsing basic if-else statements in a language like C/C++, where it is possible to write an if-else statement like this:

```
if (a) if (b) c = a + b; else c = a;
```

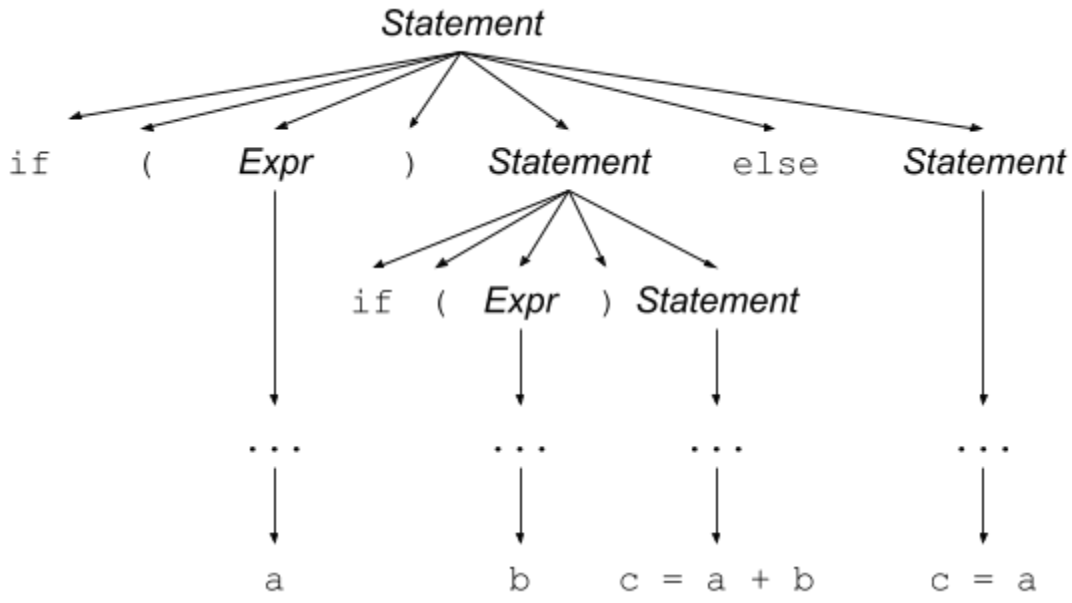
- We might initially write a CFG like the following to parse such statements:

1 *Statement* → `if (` *Expr* `)` *Statement* `else` Statement
2 | `if (` *Expr* `)` *Statement*
3 | `...`

- In this case, however, the above statement actually has two different rightmost derivations, one that associates the `else` with the first `if`, and one that associates it with the second. The former's parse tree looks like this:



- The parse tree for the derivation that associates the `else` with the second `if` looks like this:



- These two derivations would result in two different behaviors in the compiled code. The first would evaluate `c = a` only if both `a` and `b` evaluate to false. The second would evaluate `c = a` if just `a` evaluates to false, regardless of the value of `b`.

- We could remedy this ambiguity by modifying the grammar to force an `else` statement to associate with the innermost `if` statement (a standard convention in most programming languages):

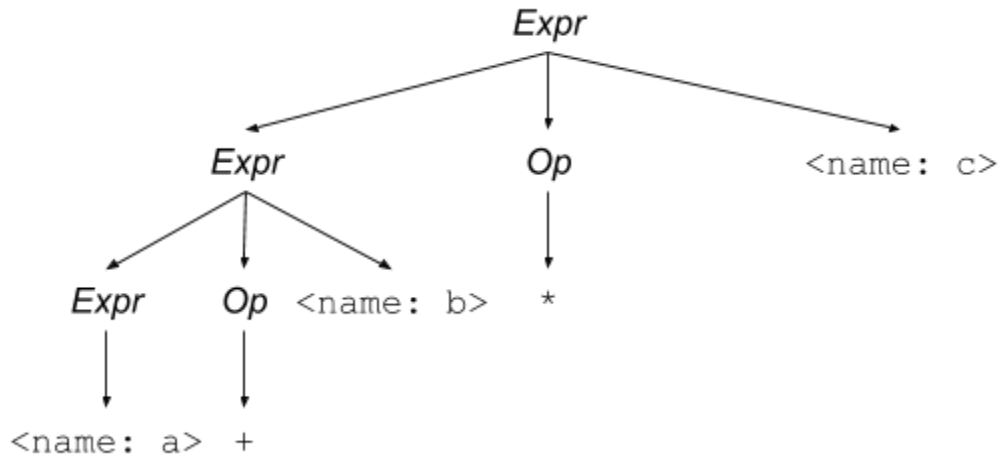
1	*Statement*	→	`if ( Expr )` *Statement*
2		|	`if ( Expr )` *WithElse* `else` Statement
3		|	`...`
4	*WithElse*	→	`if ( Expr )` *WithElse* `else` Statement
5		|	`...`

- Under this grammar, the `else` statement in our example above is forced to associate with the second `if` statement, resulting in a derivation similar to the one that produced the first parse tree above.

- In general, a compiler writer must be careful about writing ambiguous grammars, like the original attempt above at an if-else grammar. There are, however, automated/systematic ways to deal with some kinds of ambiguity.

Effects of grammar structure

- Even if our grammar is unambiguous, the way we structure the grammar can have unintended consequences. For example, consider the parse tree that would result from a derivation of the sentence `a + b * c` under the expression grammar above:



- Under a natural interpretation of this parse tree, the operation `a + b` would be executed first, and the result of that operation would be multiplied by `c` to produce a final result of `(a + b) * c`. This would violate the standard order of operations.

- This happens because the structure of our grammar treats all of the operators the same way, without any regard for precedence.

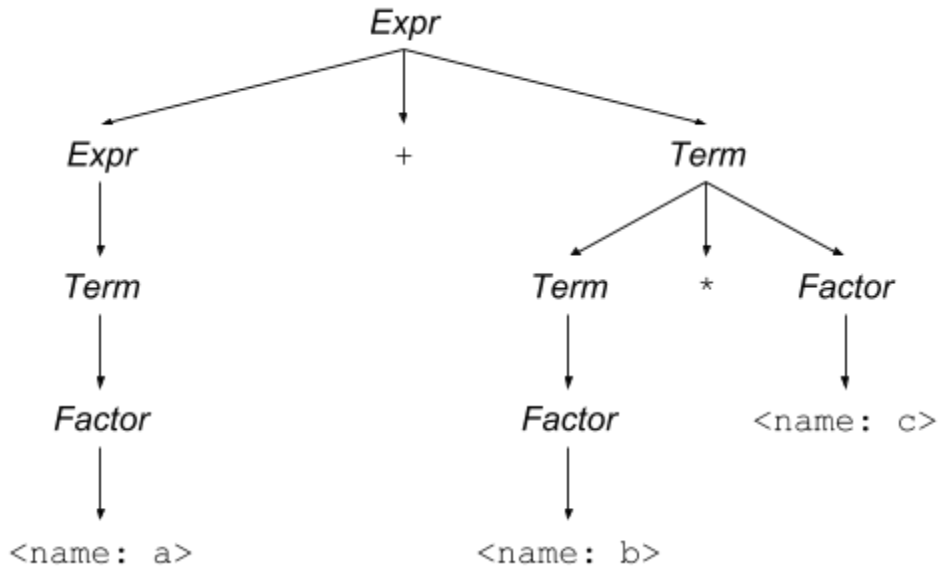
- We can fix this by encoding levels of precedence into our grammar. For example, in the expression grammar, we want `()` to have the highest precedence, `*` and `/` to have next-highest precedence, and `+` and `-` to have lowest precedence.

- The following grammar captures this precedence ordering:

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* `+` *Term*
2		|	*Expr* `-` *Term*
3		|	*Term*
4	*Term*	→	*Term* `*` *Factor*
5		|	*Term* `/` *Factor*
6		|	*Factor*
7	*Factor*	→	`(` *Expr* `)`
8		|	`num`
9		|	`name`

- The key to this grammar is that each higher level of precedence is forced to drop to a new, lower level of the parse tree in the derivation, with each *Term* dropping one level below an *Expr*, and each *Factor* dropping two levels lower. This means that the higher-priority operations must be evaluated completely before using their result in a lower-priority expression.

- We can see that this grammar fixes the parse tree for the sentence `a + b * c`:



- As an added benefit, this grammar fixes the expressivity of the expression grammar we wrote above, allowing us to write expressions like `a*(b+c)`.

- To add operators of even higher precedence (e.g. array subscripts like `a[i]`), we could continue to add further levels below *Factor***.**

Using CFGs in parsers to discover derivations

- The goal of the parser is to take the lexeme/syntactic category pairs output by the scanner and to construct a derivation (e.g. leftmost or rightmost) for the source program if one exists. In no derivation exists, the parser must generate an informative error message.

- The work of the parser can be understood as building the parse tree (or a more compact version of it known as an ***abstract syntax tree***) for the source program.

- In this understanding, both the root of the tree and its leaves are known. In particular, the root of the tree is the start symbol of the grammar, and the leaves of the tree are the annotated words output by the scanner.

- The parser's job, then, is to attempt to discover the grammatical connection between the root and leaves of this tree. There are two different approaches to performing this discovery:
    - A ***top-down parser*** starts with the root and attempts to grow the tree towards the leaves, at each step choosing a nonterminal symbol on the lower fringe of the tree and extending it downward by expanding it with some production in the grammar.
    - A ***bottom-up parser*** starts with the leaves and attempts to grow the tree towards the root, at each step choosing a contiguous substring of terminal and nonterminal symbols in the tree's upper fringe that matches the right-hand side of some production in the grammar and generating a node corresponding to the left-hand side of that production to expand the tree upwards.

- The type of parser we use depends on some characteristics our language's grammar.

- In fact, the universe of CFGs consists of several different classes of grammar. Two of these classes are of particular interest to us:
    - ***LR(1) grammars*** can be parsed efficiently with a bottom-up parser.
    - ***LL(1) grammars*** can be parsed efficiently with a top-down parser.

- These classes are especially interesting because they can both be parsed efficiently using a left-to-right scan of the output of the scanner with at most one word of lookahead.

- LL(1) grammars are a subset of the LR(1) grammars, which in turn are a subset of all CFGs.

- Almost all programming-language constructs can be expressed in LR(1) form and often in LL(1) form.