Languages (Grammars)   Carol Zander

Context-free grammars
The next category of language is context-free languages. A context-free grammar generates strings in a context-free language. Recall that all programming languages can be defined using a context-free grammar. A context-free grammar is defined with four elements:

1. Set of terminals   (characters or tokens, these are the language constants)
2. Set of non-terminals   (these are like variables)
3. Starting non-terminal
4. Set of production rules each of which have the form:
   ```
   <non-terminal> ::= anything
   ```

The restriction here is that the left-hand side of the production rule can only be a single non-terminal symbol and nothing else. This notation is Backus-Naur form (BNF), sometimes referred to as Backus-normal form. Often an arrow is used instead of ::= .

Example 1. We can create a context-free grammar for a (very) limited subset of English.

```
<sentence> ::= <noun-phrase> <verb> <noun-phrase>
<noun-phrase> ::= <noun> | <adjective> <noun-phrase>
<noun> ::= apples | bugs | cats | dogs | bears | tables
<verb> ::= eat | juggle | chase
<adjective> ::= big | small | red | wiggly | sweet | furry
```

Note that the bar, the OR operator, is commonly used, but there are really multiple rules. For example, the second rule above is equivalent to two rules:
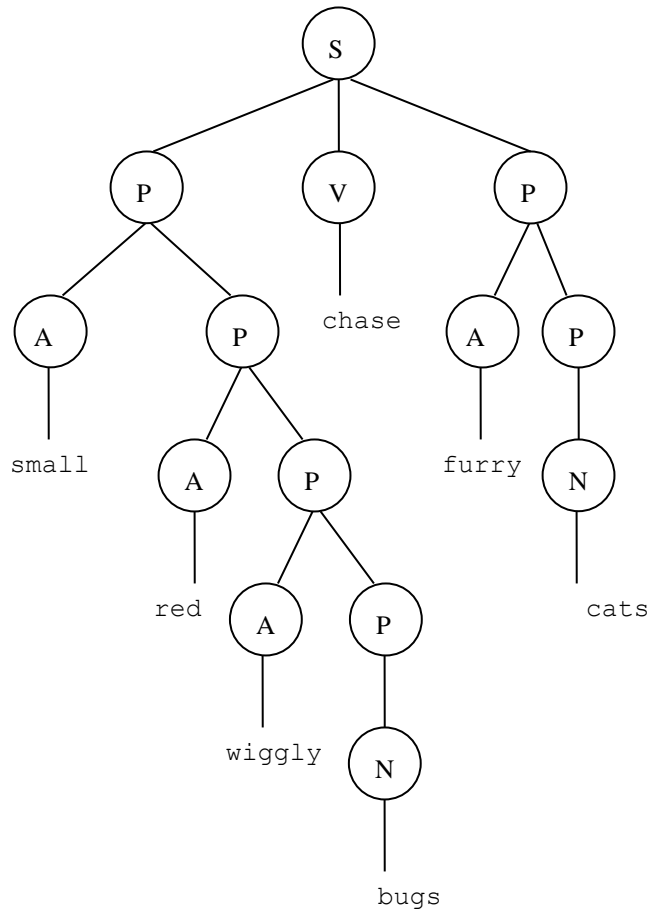
```
<noun-phrase> ::= <noun>
<noun-phrase> ::= <adjective> <noun-phrase>
```

Often only the set of production rules is given because everything is known through that. The first non-terminal, <sentence>, is the defaulted start symbol. All the other non-terminals are the left hand sides of the rules. All the terminals are any characters found in the rules.

Using the grammar, we can construct a sentence by repeatedly using the production rules to replace non-terminal symbols with other symbols (sometimes terminal, sometimes non-terminal. Let's abbreviate <sentence> to S, <noun-phrase> to P, <noun> to N, <verb> to V, and <adjective> to A. There are two ways to show how a string is derived from the grammar. One way, **a derivation,** is somewhat algebraic; the other is more visual, **a parse tree.** In a derivation, the double-edged arrow is used. Here a derivation shows one sequence of substitutions:

```
S  ==> PVP
   ==> APVP
   ==> AAPVP
   ==> AAAPVP
   ==> AAANVP
   ==> AAANVAP
   ==> AAANVAN
   ==> small red wiggly bugs chase furry cats
```
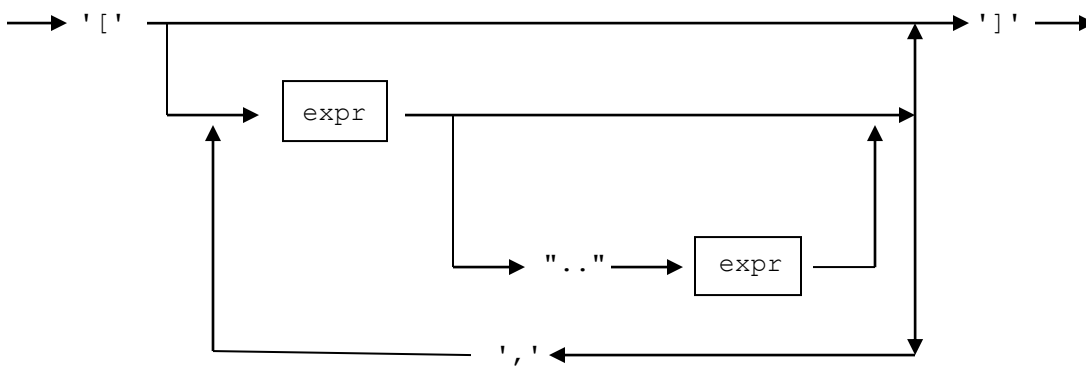
The equivalent parse tree is seen as follows:

```
                          S
             ┌────────────┼────────────┐
             P            V            P
         ┌───┴───┐        │         ┌──┴──┐
         A       P      chase       A     P
         │    ┌──┴──┐               │     │
       small  A     P             furry   N
              │  ┌──┴──┐                   │
             red A     P                 cats
                 │  ┌──┴──┐
              wiggly A    P
                     │    │
                  (empty) N
                          │
                        bugs
```

(Parse tree: S → P V P; leftmost P → A("small") P, P → A("red") P, P → A("wiggly") P, P → N("bugs"); V → "chase"; rightmost P → A("furry") P, P → N("cats"))

Example 2. See the declaration list grammar (another link).

Example 3. Consider another grammar example for a set of ints in the Pascal language. The set ADT is built right into Pascal (in a limited manner). In language books, there used to be syntax diagrams to show the syntax for a construct in a language. A terminal is in quotes, a non-terminal is in a rectangle. There would be another syntax diagram for any non-terminal used. It is essentially a graph showing the proper flow of syntax.

Syntax diagram:

```
──▶ '['──────────────────────────────▶ ']' ──▶
         ┌─▶[ expr ]──────────────────┐
         │           └──▶ ".." ──▶[ expr ]──┘
         └──────────── ',' ◀──────────────┘
```

Some examples of set values:

```
[]                        // the empty set
[2]                       // set containing only 2
[2,4,6]                   // set containing 2, 4, and 6
[1,2,5..10,15,20..30,45]  // the ".." says to include all ints in
                          //    the sequence, so the set contains
                          //    1, 2, 5, 6, 7, 8, 9, 10, 15,
                          //    20, 21, 22, 23, 24, 25, 26,
                          //    27, 28, 29, 30, and 45
```

Grammar:

```
<set-value>      -->  '[' <opt-value-list> ']'
<opt-value-list> -->  λ  | <value-list>
<value-list>     -->  <value>  | <value>  ','  <value-list>
<value>          -->  <expr>  | <expr>  ".."  <expr>
<expr>           -->  seen below
```
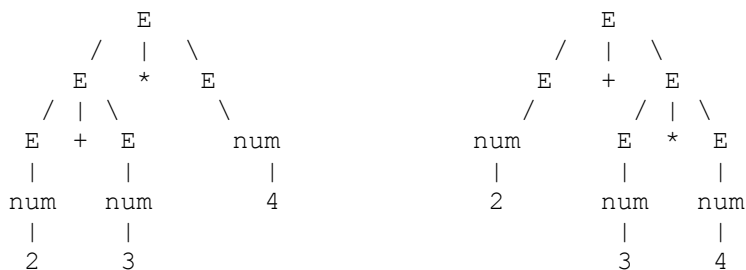
Notice the *optional* non-terminal. This is used to avoid ending the value list in a comma. If there is both a base case of λ and something else, then `<value-list>` can be empty, thus ending in a comma. The optional non-terminal avoids the problem.

Example 4. Now consider a grammar for expressions. First is a simple "bad" grammar. Then we'll look at a "good" grammar. For expressions, limit the operators to +, -, *, and /. It has the usual precedence with * and / having higher precedence than + and - . Parentheses are allowed and the expression can contain identifiers or numbers.

The *bad* expression grammar has one non-terminal, <expr>:

```
<expr>  --> <expr> + <expr> | <expr> - <expr> | <expr> * <expr>
          | <expr> / <expr> | ( <expr> ) | <<ident>> | <<number>>
```

Why is this grammar bad? Consider a string in the language such as `2+3*4`. There are two parse trees that can be built by either expanding the first or the second non-terminal:

```
        E                           E
      / | \                       / | \
    E   *   E                   E   +   E
  / | \       \               /       / | \
 E  +  E       num          num      E  *  E
 |     |        |            |        |     |
num   num       4           2        num   num
 |     |                             |      |
 2     3                             3      4
```

This is known as an **ambiguous grammar**, defined to be a grammar that can generate two distinct (differently shaped) parse trees for the same string. Compilers are not happy with ambiguity. Which parse tree do we want to generate? We want the second one. With the multiplication further down in the parse tree, its subtree has to be dealt with before dealing with the plus, which is what we want since multiplication has a higher precedence than addition.
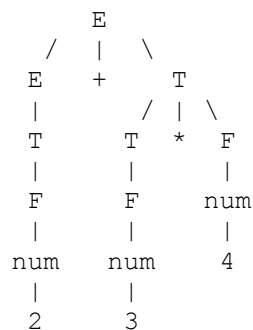
The *good* expression is grammar is not ambiguous. The precedence of operators is built right into the grammar. It uses three non-terminals for an expression, term, or factor. Notice that the higher up in the grammar, the lower the precedence of the operator. This is how operators with lower precedence get to be higher up in the parse tree. Also notice that recursion is on the left. This builds the correct parse, operation is done left to right, for expressions with operators that have the same precedence such as 2+3+4 . Operators on the left are higher up in the tree. Notice that when parentheses are used, <expr> is inside because precedence of operators is reset in the parentheses.

```
<expr>  --> <expr> + <term> | <expr> - <term> | <term>
<term>  --> <term> * <factor> | <term> / <factor> | <factor>
<factor> --> ( <expr> ) | <<ident>> | <<number>>
```

The parse tree built is the shape of the second tree above. E, T, and F are used for <expr>, <term>, <factor>.

```
          E
        / | \
      E   +   T
      |     / | \
      T    T  *  F
      |    |     |
      F    F    num
      |    |     |
     num  num    4
      |    |
      2    3
```

Example 5. The non-regular language: { $a^n b^n$ | integer $n \geq 0$} = {λ, ab, aabb, aaabbb, ...}
The grammar is straightforward and simple, although somewhat different than grammars we've looked at previously in that strings grow in the middle.

```
<S>  --> λ | a <S> b
```

Context-sensitive grammars
Context-sensitive grammars allow production rules in which the context of how the left hand side is used in the right hand side matters. This means that the left hand side of a production rule can have more than one symbol, including both terminals and non-terminals.

An example of a context-sensitive grammar is for the language L = {$a^n b^n c^n$ | $n \geq 0$} with alphabet {a, b, c}. This language is not context free.
```
S  ::= aSBC | λ
CB ::= XB
XB ::= XC
XC ::= BC
aB ::= ab
bB ::= bb
bC ::= bc
cC ::= cc
```