4.0 BNF (Backus-Naur Form) and Parse Trees

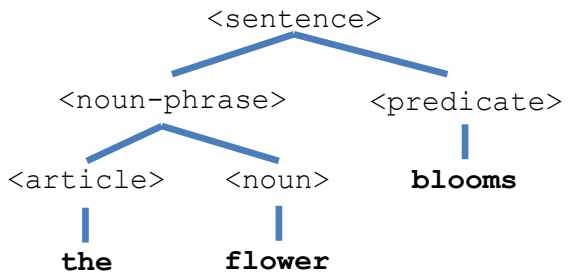
Consider again the simple grammar shown previously in section 2.0 –

```
<sentence>    ::= <noun-phrase> <predicate>
<noun-phrase> ::= <article> <noun>
<article>     ::= the | a | an
<noun>        ::= cat | flower
<predicate>   ::= jumps | blooms
```

We showed how sentences can be derived by a series of replacements:

```
<sentence>
→ <noun-phrase> <predicate>
→ <article> <noun> <predicate>
→ the flower blooms
```

Usually, derivations are more useful if they are done as parse trees.
The same derivation of "the flower blooms", expressed as a parse tree, is:



Some things to notice about Parse Trees:

- the start symbol is always at the root of the tree,
- every leaf node is a terminal,
- every interior node is a non-terminal, and
- the sentence appears in a left-to-right traversal of the leaves.

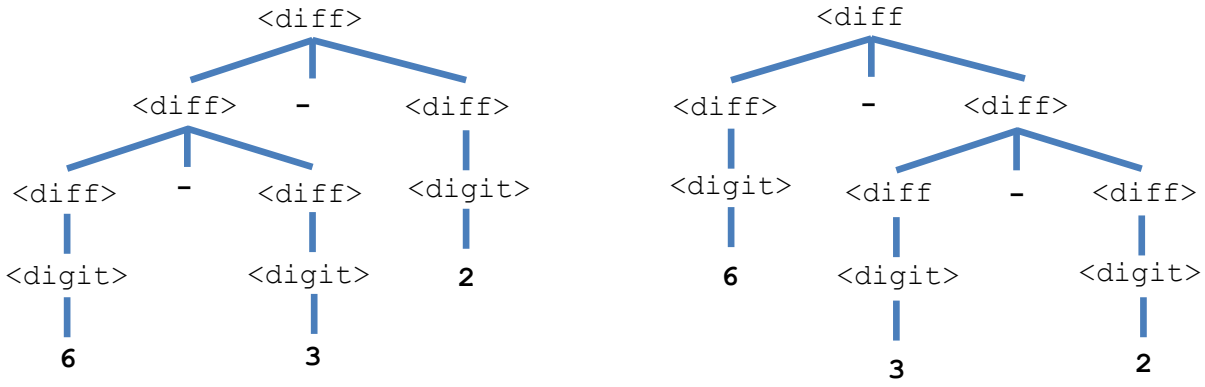
When using BNF to specify a programming language, the non-terminals of the grammar are comprised of the tokens produced by the lexical scanner.

Therefore, Parse Trees reveal the *syntactic structure* of the sequence of tokens that make up a computer program. It is very important that the grammar not be ambiguous. If the grammar were ambiguous, programs wouldn't be portable because there would be more than one way that compilers could translate them. Here's an example:

Consider the following BNF grammar for subtraction of digits:

```
<diff>  ::= <diff> - <diff> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Notice that, using this grammar, there are *two* parse trees for `6-3-2`



The left tree implies a result of `(6-3)-2` which is `1`.
The right tree implies a result of `6-(3-2)` which is `5`.
(see how the parse tree already reveals some of the semantics!)

Thus the above specification is ambiguous, and therefore is an inadequate specification of the subtraction operator. The solution is to write it so that it limits recursion to one side or the other, but not both, as follows:

```
<diff>  ::= <diff> - <digit> | <digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

The above example illustrates the use of BNF to specify *left-associative* operators (such as +, -, *, /, etc.). There are also *right-associative* operators such as the exponent (^). Can you see how that would be written?

Here is an unambiguous grammar for mathematical expressions. It has both left-associative and right-associative operators. It uses different levels of non-terminals to express operator precedence, and also parentheses so that the programmer can use nesting and override precedence:

```
<exp>    ::= <exp> + <term> | <exp> - <term> | <term>
<term>   ::= <term> * <power> | <term> / <power> | <power>
<power>  ::= <factor> ^ <power> | <factor>
<factor> ::= ( <exp> ) | <int>
<int>    ::= <digit> <int> | <digit>
<digit>  ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```