Review: Right/Left-Most Derivations

- A **right-most** derivation is one in which we always choose the nonterminal symbol furthest to the right for substitution.
- A **left-most** derivation does the same, always on the furthest left.

For example, in this basic integer grammar, we can derive 123 in two ways:

\[
\begin{align*}
\text{integer} & \rightarrow \text{digit} | \text{digit integer} \\
\text{digit} & \rightarrow 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 \\
\text{expr} & \rightarrow \text{id} | \text{number} | \text{expr} \text{ op} \text{ expr} \\
\text{op} & \rightarrow + | - | * | / \\
\end{align*}
\]

Note: in this language 'id' and 'number' are shorthand for strings of characters/integers, respectively.

The last line compresses multiple steps of further derivation that would be needed to go from each occurrence of 'id' to the final actual variable identifiers.

Deriving and Parsing

Given this grammar, we can derive, e.g., the expression "x * y - z":

\[
\begin{align*}
\text{expr} & \rightarrow \text{id} | \text{number} | \text{expr} \text{ op} \text{ expr} \\
\text{op} & \rightarrow + | - | * | / \\
\end{align*}
\]

Note: in this language ‘id’ and ‘number’ are shorthand for strings of characters/integers, respectively.

The last line compresses multiple steps of further derivation that would be needed to go from each occurrence of ‘id’ to the final actual variable identifiers.
Deriving and Parsing

- One technical detail: this derivation is neither right-most nor left-most
  - If id "number" are actually non-terminals, we need to replace them as we go along
- In practice, however, any such 'pseudo'-derivation can be replaced with one that is of the strictly proper form
  - Going forward, when we want to focus on expression structure, we ignore this detail

```
expr → expr op expr
expr → expr op id
expr → expr + id
expr → expr op expr - id
expr → expr op expr + id
expr → id * id - id
expr → x * y - z
```

Parse Trees

- Derivations of grammatical strings can also be regarded as parse trees
  - The root of such a tree is the starting non-terminal
  - Every leaf is a terminal symbol
  - Every internal node consists of the result of a production rule
  - For example, for our basic integer grammar, the derivation of 415 shown has a corresponding tree:

```
integer → digit | digit integer
digit → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Parsing and Language Structure

```
expr → id | number | - expr | (expr)
expr → expr op expr
op → + | - | * | / 
```

- In this (ambiguous) grammar, the given (pseudo-)right-most derivation corresponds to a particular parse tree, from which the precedence structure can be read

```
expr → expr op expr
expr → expr op id
expr → expr + id
expr → expr op expr - id
expr → expr op expr + id
expr → id * id - id
expr → x * y - z
```

Typically, we understand a complete expression in a language like this as anything corresponding to a subtree of the parse tree

- It is often possible to construct a parsing algorithm recursively on such a basis

For this tree, 'x * y' is a complete expression (left-hand subtree), and 'z' is another (right-hand).

Thus, this parse tree implies that we have the precedence structure:

```
(x * y) - z
```

Ambiguity in Parsing

Given this same grammar, a distinctly different (pseudo-)right-most derivation is possible, corresponding to a different parse tree.

\[
\begin{align*}
expr & \rightarrow id \mid number \mid -expr \mid (expr) \\
& \mid expr \ op \ expr \\
op & \rightarrow + \mid - \mid * \mid /
\end{align*}
\]

Each of the two parse trees corresponds to a functionally different reading of the derived expression, changing the naturally corresponding order of operations.

Unambiguous Grammars

One approach to a problem like this is to add verbiage to our language specification: include edge-cases and special rules for use of the grammar in the larger, informal documentation.

A more elegant approach is a grammar without ambiguity.

Such a grammar for arithmetic expressions (including parentheses) is:

\[
\begin{align*}
expr & \rightarrow term \mid expr \ add_op \ term \\
term & \rightarrow factor \mid term \ mult_op \ factor \\
factor & \rightarrow id \mid number \mid -factor \mid (expr) \\
add_op & \rightarrow + \mid - \\
mult_op & \rightarrow * \mid /
\end{align*}
\]

Exercise: use this grammar to derive/parse

\[(x \ast y) - z\]

\[x \ast (y - z)\]
Grammar and Structure

A properly written formal grammar can impose the rules for operator precedence and associativity.
These rules are very important when we get to operation semantics.
They determine the order in which we will perform the operations.

Grammar and Precedence

These two grammars impose two different precedence orderings on the operators ‘!’ and ‘?’.

Equivalence:

- Equivalent to: 
  id ! (id ? id)
- Equivalent to: 
  (id ! id) ? id

Exercise: in each grammar, give a parse tree for:

id ! id ? id

Grammar and Structure

Here, (with expr as start state), mult_op occurs deeper in the grammar than add_op.
That is, we need more steps of a derivation to introduce mult_op.
Subexpressions using mult_op will be subtrees of any that use add_op, and will be parsed as complete on their own.
This means that mult_op has higher precedence than add_op.

Grammar and Structure

Here, both expr and term recursively occur on the left of the operators they introduce.
This means that mult_op and add_op are both left associative.

Exercise: give a parse tree for:
x + y + z

A properly written formal grammar can impose the rules for operator precedence and associativity.
These rules are very important when we get to operation semantics.
They determine the order in which we will perform the operations.
Grammars and Associativity

Since this grammar makes `add_op` left associative, the parse tree indicates that the expression:

\[ x + y + z \]

is interpreted as:

\[ (x + y) + z \]