Syntax (II)

EBNF

- To increase the clarity and brevity of syntax descriptions, Extended BNF (EBNF) was introduced.
- Based on BNF, EBNF has a few more metasymbols other than those from BNF:
  - Curly braces {}: include the enclosed symbols 0 or more times
  - Parentheses (): include the enclosed symbols 1 or more times
  - Square brackets []: indicate an optional sequence of symbols
- **Example**: Describe the integer formally in EBNF

\[
Integer \rightarrow \text{Digit} \{\text{Digit}\}
\]

- The above production cannot avoid the situation where an integer with multiple digits starts with 0. How to rewrite the production to avoid the situation?

\[
\begin{align*}
\text{Digit} &\rightarrow 0 | \ldots | 9 \\
\text{nonzeroDigit} &\rightarrow 1 | \ldots | 9 \\
\text{Integer} &\rightarrow \text{Digit} | \text{nonzeroDigit}(\text{Digit})
\end{align*}
\]

or

\[
\begin{align*}
\text{zero} &\rightarrow 0 \\
\text{nonzeroDigit} &\rightarrow 1 | \ldots | 9 \\
\text{Integer} &\rightarrow \text{zero} | \text{nonzeroDigit}\{[\text{zero}, \text{nonzeroDigit}]\}
\end{align*}
\]

Derivation

- To determine if a string is valid according to a grammar.
- A derivation is a series of replacements defined by the productions.

- Derivation steps
  - The derivation begins from the start symbol.
  - The production rules iteratively replace nonterminal symbols with nonterminal and terminal symbols until there are no more nonterminal symbols in the string.
  - In each iteration, apply a production rule to the leftmost symbol.
  - If the terminal symbols match the string, the string is valid.

- **Example**: Given the formal definition of integer, check whether 312 is a valid integer.

\[
\begin{align*}
\text{Integer} &\rightarrow \text{Digit} | \text{DigitInteger} \\
\text{Digit} &\rightarrow 0 | \ldots | 9
\end{align*}
\]
**Syntax**

**Parse Tree**
- A **graphical form** of derivation.
- Each derivation step corresponds to a new subtree.
- **Example**: Draw a top-down parse tree for 312 by using the following rules

```
Integer ⇒ Digit|DigitInteger
  ⇒ Digit
  ⇒ 3
  ⇒ Digit Integer
  ⇒ 31
  ⇒ Digit
  ⇒ 31 Digit
  ⇒ 312
```

**Operator Associativity**
- The order of non-terminal symbols on the right hand side of productions is important.
- Different orders will generate parse trees with different associativities.

- **Example 1**: Draw a parse tree for 3 - 1 + 2 using the following rules

```
Expr → Expr + Term|Expr - Term|Term
Term → Integer
Integer → Digit{Digit}
Digit → 0|⋯|9
```

```
Expr
  +
  -
  +
  -
  +
  -
  +
  -

Expr
Term
Integer
Digit

Digit

Digit

Digit

Digit

Digit

Digit

(3 - 1) + 2
```
- **Example 2**: Draw a parse tree for $3 - 1 + 2$ using the following rules

\[
\begin{align*}
  &Expr \to Term + Expr|Term - Expr|Term \\
  &Term \to Integer \\
  &Integer \to Digit\{Digit\} \\
  &Digit \to 0|\cdots|9
\end{align*}
\]

```
  Expr  \\
   \_   \\
  Term  Expr  \\
  |   |   \\
 Integer Term + Expr  \\
|   |   |   |   \\
 Digit Integer Term  \\
   |   |   \\
 Digit Integer  \\
   |   \\
 Digit  \\
   \\
 3  \\
   \\
 1  \\
   \\
 2  \\
   \\
 3 - (1 + 2)
```

- In the first $3 - 1 + 2$ example, the result is interpreted as $(3 - 1) + 2$ using depth-first traverse.
- In the second $3 - 1 + 3$ example, the result is interpreted as $3 - (1 + 2)$. They have different associativities, and the second one is not the associativity it should be.
- Maintaining the order of operations is essential in programming languages.
- In order to specify hierarchies of operations, grammars can become extremely complex.