Let's program a calculator:

```
Welcome to SeoulTech SuperCalculator!
> 3 * (5 + 7 * 2) + 30 * 2 / 15
==> 61
> 110 - (23 + 12) * (15 - 12)
==> 5
```

Tokenization (Lexical Analysis)

Tokenization means to partition the input string or text file into tokens (smallest meaningful units) such as numbers, identifiers, and operators.

```
(abc12+27 * 23.0(12abc34
```

Symbol: (
Identifier: `abc12`
Symbol: +
Number: 27.0
Symbol: $*$
Number: 23.0
Symbol: (
Number: 12.0
Identifier: `abc34`
Stop.

Whitespace (spaces, line feeds, tabs) is already removed by tokenization.

Note: Tokenizer knows nothing about the syntax of expressions or the programming language.

Tokens

We need four kinds of tokens:

- Number constants, such as 12 or 34.56;
- Variable names ("identifiers"), such as `abc12`;
- Operators (usually one-letter), such as `+`, `*`, or `(`;
- a stop token (end of input).

We use the following rules:

- Whitespace is skipped;
- A number is a string of digits with possibly a decimal point;
- an identifier starts with a letter or '_', and consists of letters, digits, and underscores;
- anything else is a one-letter symbol token.

Recursive descent parsing

An expression is a sum (with + or −) of terms.
A term is a product (with $*$ or /) of items.
An item is either a number, or a variable name, or an expression enclosed in parentheses.

For each syntactical element (that is, "expression", "term", and "item") we write a method to parse it.

Since `parse_expression` calls `parse_term`, `parse_term` calls `parse_item`, and `parse_item` may call `parse_expression`, recursive descent parsing automatically leads to indirect recursion.