Implementing a calculator

Tokenization and parsing

How does a calculator get from an arithmetic expression (which is just a string) to the value of the expression?

This question is a very simple version of what happens in interpreters or compilers for programming languages (e.g. the Javascript interpreter built into your web browser, the macro languages of Excel, Maple, or Matlab, or the Python interpreter itself).

In general, this process consists of three steps: *lexical analysis*, *syntactic analysis* and *semantic analysis*.

Lexical analysis

Lexical analysis (*tokenization*) splits the input text (a string or a text file) into minimal meaningful units. These minimal units are called *tokens*, and lexical analyzers are also called *tokenizers*.

For our calculator we need four kinds of tokens:

- numbers (real numbers, such as 3, 71, 23.45),
- identifiers (variable names, like `i`, `j`, `a123`),
- operators (`+`, `-`, `*`, `/`, $\cdots$),
- a stop token to denote the end of the input.

The tokenizer does not know anything about the meaning or syntax of expressions. It uses white space (space and tab characters and the end of line) to determine where tokens end or begin. Since white space has no meaning in the expression, no tokens are generated for white space. (Similarly, in the Python interpreter comments are removed by the tokenizer.)

For example, `123456` and `123 456` result in different token sequences, and so do `a12` and `a 12`. On the other hand, both `1+2` and `1 + 2` result in the same sequence of tokens (namely number 1.0, symbol `+`, number 2.0).

For instance, given the expression below

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

the tokenizer produces the following sequence:

```
Token: Symbol(()
Token: Identifier(abc12)
Token: Symbol(+)
Token: Number(27.0000)
Token: Symbol(*)
Token: Number(23.0000)
Token: Symbol(()
Token: Number(12.0000)
Token: Identifier(abc34)
Token: Stop
```

Note again that even though the string is nonsense (or at least it is not a meaningful expression), the tokenizer has no difficulty to split it into tokens according to its rules. The tokenizer has no understanding of syntax or meaning of the string.

Syntactic analysis

During syntactic analysis (*parsing*), the sequence of tokens produced by the tokenizer is examined and organized into a meaningful structure. Syntax errors are detected during parsing. A syntactic analyzer is also called a *parser*.

Syntactic analysis often organizes the tokens into a tree structure called *syntax tree* or *parse tree*. In our case, the tree represents the structure of the input expression. For instance, this expression

```
( 1 - 2 ) * 3.0 + 4 / a12
```

is parsed into the syntax tree of Fig. 1. Parsing is responsible for handling the priorities of the different operators: Bonding of `*` and `/` is stronger than that of `+` and `-`.

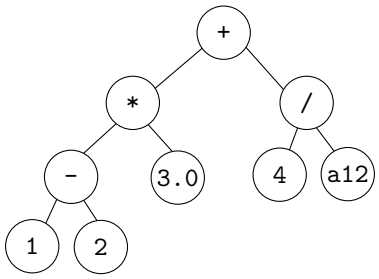

Figure 1: A syntax tree

Semantic analysis

Semantic analysis is the general term for assigning *meaning* to the parse tree produced by the parser. In our calculator, there is little semantic analysis, we simply compute the value of the expression. In a compiler, semantic analysis will involve type checking, checking the scopes of variables, and finally the compiler will generate code.

Recursive descent parsing

The technique we use for building our parser is called *recursive descent parsing*. The idea is to first draw a syntax diagram (Fig. 2) of the expressions we want to parse.

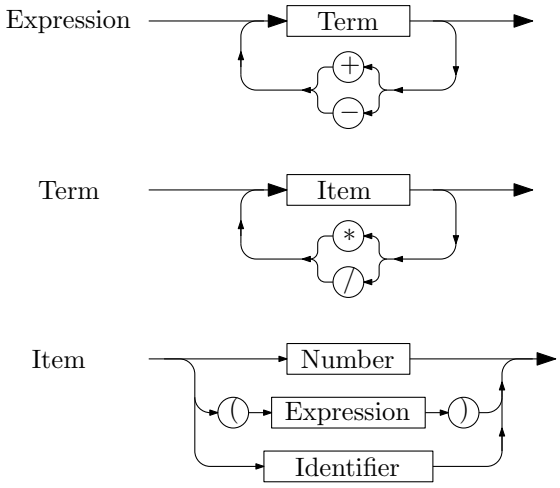

Figure 2: Syntax diagram of simple expressions

The syntax diagram explains the three different syntactic elements that can appear in an expression: *Expression*, *Term*, and *Factor*. An "Expression" is a sequence of "Terms" separated by addition and subtraction. A "Term" is a sequence of "Items" separated by multiplication and division. Finally, an "Item" is either a number, an identifier (that is, a variable name), or an arbitrary expression in parentheses.

To parse a given expression, all we have to do is to follow the arrows in the syntax diagram. It suffices to look at the next token in the token sequence to determine which branch to take at each step.

Once we have the syntax diagram, the implementation is very easy. We write one method for each syntactic element: `parse_expression`, `parse_term`, and `parse_item`. Each method is responsible to parse one kind of syntactic element. It returns the value of the element (a number), and throws an exception if a parsing error occurs.

As an example, this is the code for `parse_term` (see example code in *calculator1.py*):

```
def parse_expression(tok):
  result = parse_term(tok)
  t = tok[0]
  while t.isSymbol("+") or t.isSymbol("-"):
    tok.pop(0)
    rhs = parse_term(tok)
    if t.isSymbol("+"): result = result + rhs
    else: result = result - rhs
    t = tok[0]
  return result
```

Extending the parser

The simple syntax diagram above does not contain a unary minus operator, and so we cannot write negative numbers or expressions like `-a` or `-(3 + 5)`. Furthermore, we would like to add the operator `^` for exponentiation.

We have to be a bit careful here: While `-` and `/` are *left-associative* (meaning that $a - b - c - d = ((a - b) - c) - d$), exponentiation is *right-associative* (in mathematics, $2^{3^2} = 2^9 = 512$, and similar we would like `2^3^2 = 2^(3^2) = 2^9 = 512`).

The syntax diagram in Fig. 3 correctly implements unary plus and minus and a right-associative exponentiation operator. You can find the implementation of this syntax diagram in the example code in *calculator2.py*.

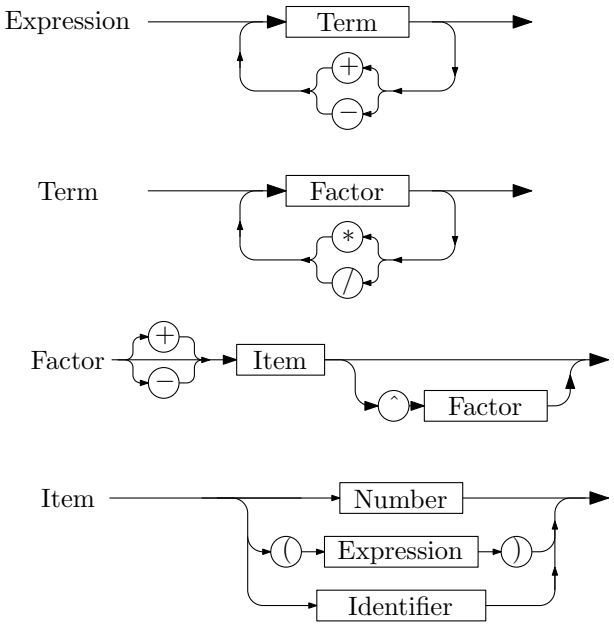

Figure 3: Extended syntax diagram with power and unary operators