C++ Design project: creating a calculator

Numerical geology option, ENSG 2023-2024

Guillaume Caumon, Thierry Valentin, Sophie Viseur.

Translated by Amandine Fratani

1 Introduction

A calculator can be seen as a black box (Fig. 1) that takes as input a character string corresponding to an operation (i.e., a mathematical expression) and sends as output the result (number) of this operation.

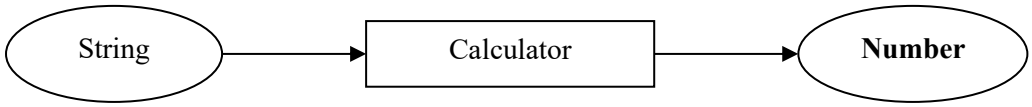


Figure 1: *Diagram of a calculator, input and output.*

In order to calculate this result, the calculator needs to understand the structure and meaning of the expression given to it in the form of a character string. It is therefore important to be able to define which types of expressions (sentences) can be given as input and according to which rules. This operation is equivalent to establishing a language.

The first part will cover concepts of computer language and related concepts. In the second part, these concepts will be applied to the specific case of setting up a programme to create a calculator.

2 Some notions about language, grammar and compilation

2.1 Languages: vocabulary, syntax and semantic

A language is composed of a **vocabulary** (words made up of characters from an **alphabet**), a **grammar** and **semantics**. For example, "caat" is not a word in the English vocabulary (lexical error). "The beach mouse the cat" is made up of valid words but is not a grammatically correct sentence (syntactical error). “The cat drinks a boat” is a grammatically correct but semantically incorrect (or at least obscure) sentence.

A computer language is a formalism for representing and manipulating information. It follows the same rules as common languages such as English: it consists of an alphabet (i, +, *) forming a vocabulary (for, i while, 3.5), used to write sentences (int i=2;) that carry a certain amount of information (semantics).

Three aspects are often distinguished when analysing a sentence in a language: the lexical, syntactic and semantic aspects.

2.1.1 lexical analysis:

It cuts the input sentence into a sequence of words (lexical unit) from the considered vocabulary. The purpose is to check whether the characters used belong to the language alphabet and whether they form words authorised by the language.

Example of a lexical error:

- 2.5.5 in mathematics and C++
- “arbirtrary” in English
- Warning: 2,5 may be a lexical error if your vocabulary stipulates that the comma is specified by a full stop '.' (English convention) but will be valid with the French convention.

2.1.2 Syntactic analysis:

Grammar defines the syntax of a sentence. It sets out the laws regarding the possible orderings of words in the vocabulary. Syntactic analysis therefore checks that sequences of lexical units respect a sentence structure imposed by the **grammar** of the language and rejects invalid constructions.

Example of a syntax error:

- a missing parenthesis in a program
- "3++2=5" is an incorrect arithmetic expression.

Syntax is a determining element of the language, for example: "x++" is not acceptable according to arithmetic syntax, but acceptable for the syntax of a computer language such as C++.

2.1.3 Semantic analysis:

It checks the meaning of sentences in language.

Examples:

- “int i = 2.5;” is a semantic mistake
- Warning!!!
  “if(2+4 == 1) cout<<"Hello"<<endl;” is semantically correct in C++, the fact that 2+4 is not equal to 1 does not affect the semantics of the C++ language.
  On the other hand: “"the cat" == true” is semantically false because the C++ language does not allow a Boolean to be compared with a character string

NB: According to language complexity and nature, some part of the sentence analysis will be more or less dominant.

2.2 Some remarks on informatic language

Programming languages play an essential role in managing complexity. In informatics, as in mathematics, managing complexity requires abstraction and conceptualization of the problems to be solved. Programming languages makes it possible to define conceptual logic bricks that are easier to manipulate than the elementary, low-level operations available on a computer. In a way, the evolution of languages can be seen as a continuous effort of abstraction aimed at escaping the constraints of machine structure and to forge the tools to write programs whose structure reflects that of the problems to be solved and not that of the machines used. In this way, computer scientists have brought the languages they use closer to mathematics and logic, even if the short history of computing has not yet enabled us to achieve the level of homogeneity and fluidity that mathematical language has reached after several millennia of existence.

Finally, the use of sufficiently abstract languages facilitates the writing of programs, their comprehension by people other than programmers and their assembly into complex software. It also opens up the possibility of reasoning about programs, of defining their semantics and performing proofs of correctness.

Two styles of language are commonly used: interpreted languages rely on an interpreter to translate the language instructions into instructions for the processor. This translation takes place as the program is executed (e.g. python, caml, prolog, javascript...).

To save time, it is also possible to use a compiled language, in which the processor instructions are generated by a compiler before the program is executed. This is known as object code. The languages concerned are Fortran, C and C++.

In the case of a calculator, the language created (whose phrases correspond to all the mathematical expressions accepted by the calculator) is relatively simple. An **interpretation** is therefore sufficient.

2.3 Formal grammar

Formal grammars form the basis of text analysis techniques and are used to solve a variety of pattern recognition problems. Several types of formal grammars have therefore been developed to address different recognition problems.

In the case of a calculator, the grammar is used to define which mathematical expressions (constructed from the vocabulary under consideration) are permitted (with which syntax).

A G grammar is a quadruplet: **G(terminals, non-terminals, productions, axiom)**, where[1] :

- terminals: set of words forming sentences that cannot be replaced;
- non-terminal: transitional parts of sentences that correspond to non-terminal concepts and can therefore be replaced by production rules;
- axiom: one of the non-terminal elements that is the root (i.e., the basic expression) of the grammar;
- productions: grammatical rules that define the set of terminals and non-terminals by which a non-terminal can be replaced. Rules have the form head ➔ body, where "head" is a non-terminal and "body" can be sequences of terminals and non-terminals. Each production allows you to rewrite a sentence by replacing the term on the left with the term(s) on the right. For example, the rule Expr ➔ Expr + Expr | Expr – Expr | Number can be used to define an elementary grammar to describe the structure of sentences such as 2+1-4+5-4.

2.3.1 Grammar visualization: syntax and semantics tree

The *syntax tree* (Fig. 2) corresponds to the tree built by parsing. The root of this tree corresponds to the grammar axiom and its leaves corresponds to terminals. Intermediate branches represent non-terminals.

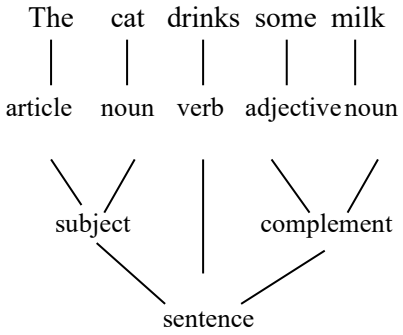


Figure 2: *Syntax tree for the sentence " the cat drinks some milk ".*

However, for reasons of efficiency, it is often not a syntactic tree that is built, but a *semantic tree* (Fig. 3). The semantic tree can be seen as a syntactic tree in which only the elements that make sense of the sentence have been retained.

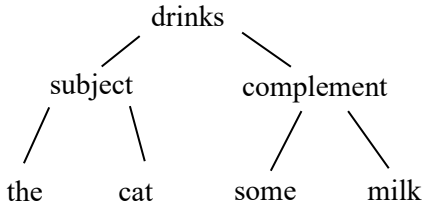
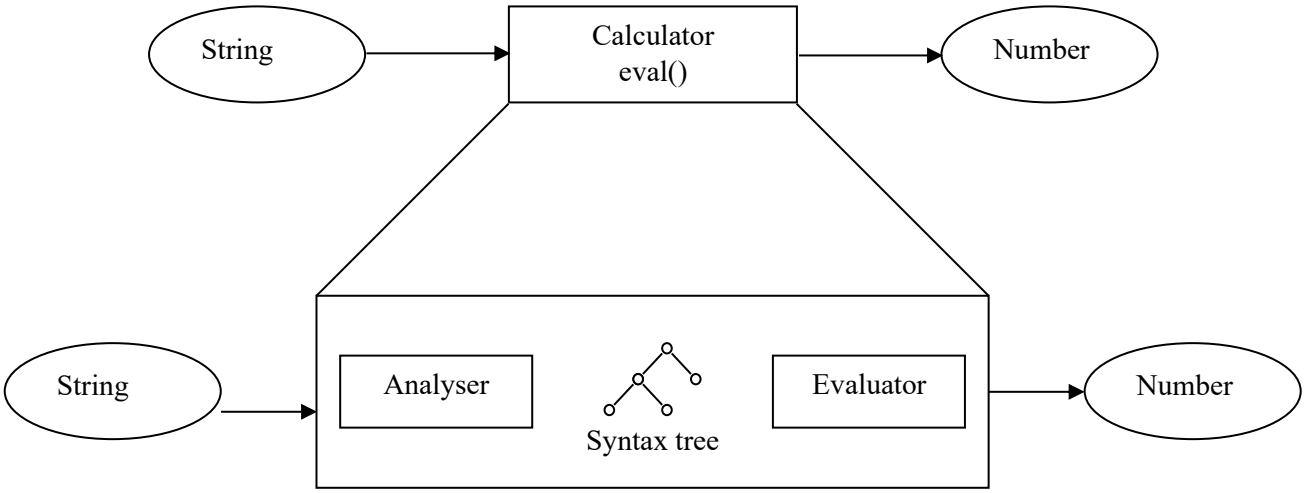


Figure 3: *Semantic tree corresponding to the phrase "the cat drinks some milk"*

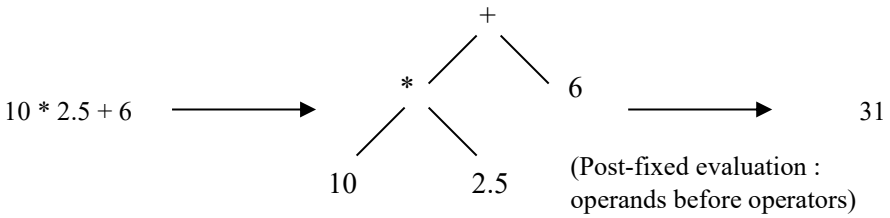
[1] http://fr.wikipedia.org/wiki/Grammaire_formelle

3 Calculator conception and design

The first step in evaluating an expression is to analyse the string, which generates a tree. This can then be passed to the evaluator:

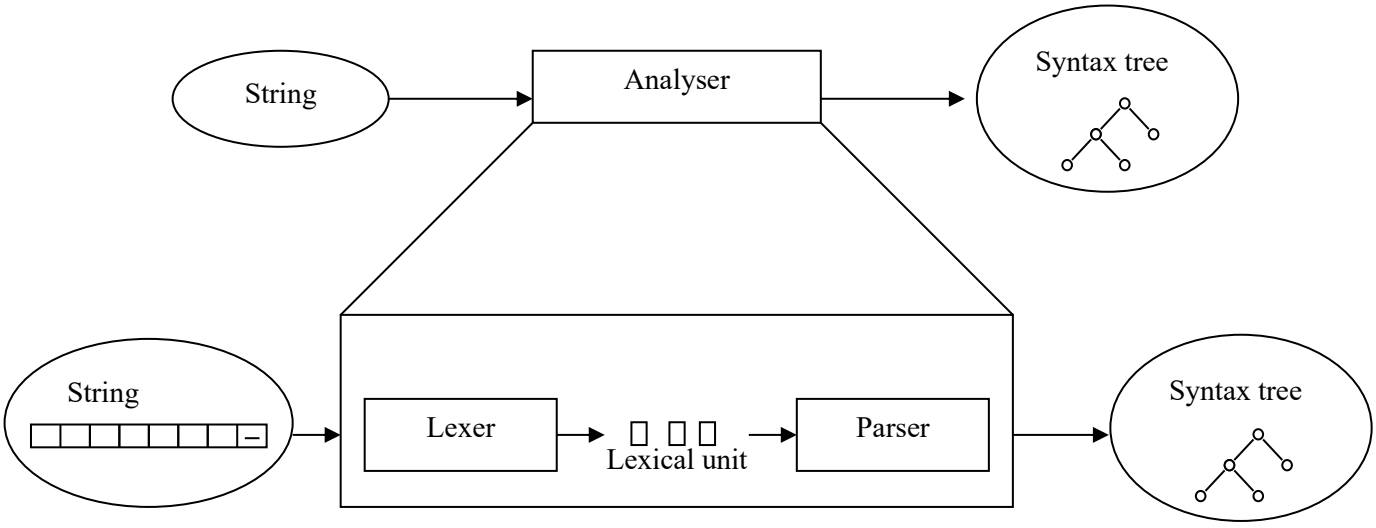


Example : (semantic tree)

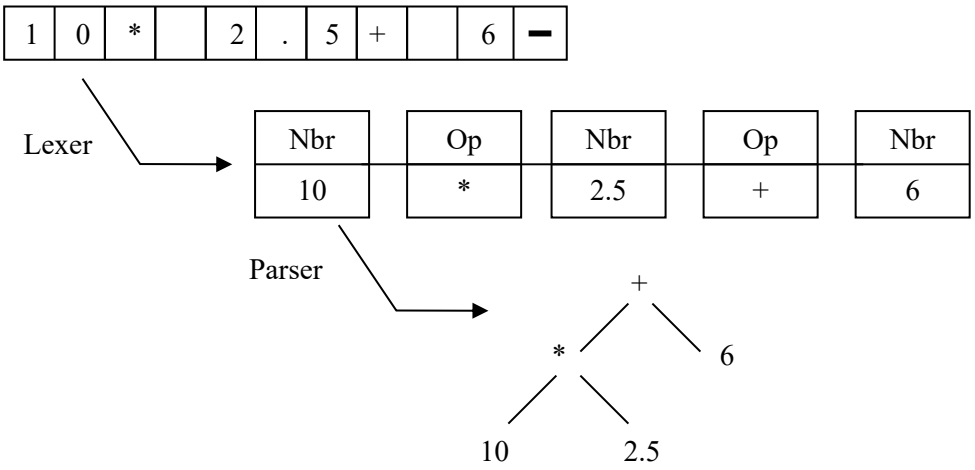


The analyser itself can be decomposed into:

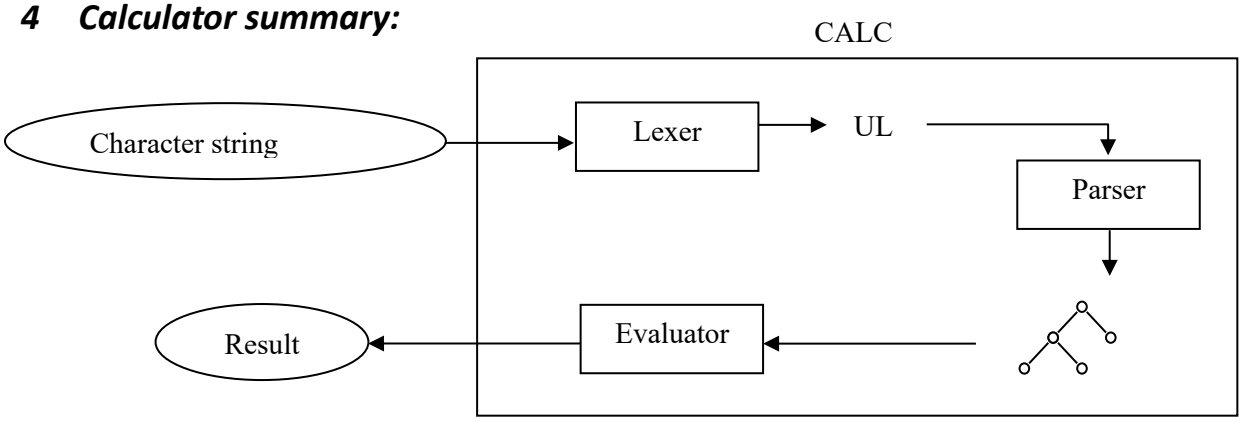
- a lexical analyser (lexer): which divides the character string into lexical units (words);
- a parser: produces the syntax tree from words and their grammatical meaning.



Example :



4 Calculator summary:



Collaboration diagram: