



# B1 - Unix & C Lab Seminar

---

B-CPE-101

## Bootstrap

---

EvalExpr



3.0



# Bootstrap

language: C



- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

The goal of the *EvalExpr* project is to help you understand the basis of parsing (syntax analysis). Mathematical expressions analysis is quite simple; the main difficulty lies in managing precedence (parentheses are prioritized over products, and products are prioritized over additions).

There are three representations of mathematical expressions:

- **Infix Notation:** operators are written between the operands they operate on. For instance,  $21 + 42$ .
- **Prefix Notation:** operators are written before operands. For instance,  $+ 21 42$ .
- **Postfix Notation:** operators are written after operands. For instance,  $21 42 +$ .

The **infix notation** operator is the most common. However, it is the most complex notation for a computer to understand.

There are many algorithms such as, *Shunting-yard*, written by Edsger Dijkstra, that converts infix notations into the two other ones.

## PART 1: RECURSIVE DESCENT PARSING

Recursive descent parsing is a popular approach to the preceding problem. Like its name suggests, it is a parser that is composed of mutually recursive functions.

Each function implements a specific type of operation.

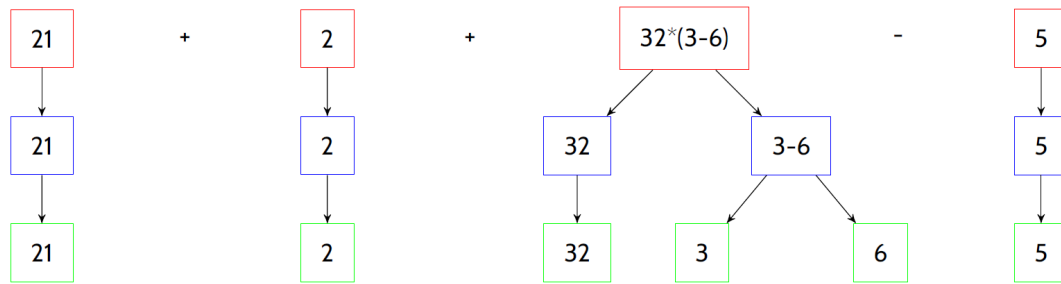
Technically speaking, it is a predictive parser, and belongs to the **LL(k)** grammar category.

The goal is to view an expression like you would a hierarchical structure, with a set of summands on the top level. Summands are separated by '+' and '-' signs.

Each summand consists of a single number or of several factors. Factors are separated by '\*', '/' and '%'.

Factors can be whether a single number or a **sub-expression in parentheses**. Those sub-expressions can be parsed like whole expressions.

For instance, here is how  $21 + 2 + 32 * (3 - 6) - 5$  gets parsed:



<https://en.wiktionary.org/wiki/summand>

## PART 2: NUMBER

Write a function that converts the beginning of a string, given as parameter, into an integer. The function should return the previously converted integer, and must be prototyped the following way:

```
int number(char **str_ptr);
```

As with `strtol`, you should move your string's pointer, given as parameter, to the end of the number (or the first invalid character).

You can also use a second parameter to accomplish this.

```
int my_strtol(char *str, char **endptr);
```



`man strtol`

This function is the key to dealing with parentheses, signs before parentheses, ...



## PART 3: SUMMANDS

---

Write a function named `summands` that returns the sum of the expression given as parameter.

```
int summands(char **str_ptr);
```



Call your previous function in order to handle each summand separately.



Start by handling one number, 42, then two, 42+12 or 42-21, eventually leading to infinity.

## PART 4: FACTORS

---

Write a function named `factors` that returns the product of the expression given as parameter.

```
int factors(char **str_ptr);
```



Call the `number` function in order to handle each factor separately.



Start by handling one number, 42, then two, 42\*12 or 42/21, eventually leading to infinity.



## CONCLUSION

---

Given that an expression is a set of additions of products of numbers, you should modify your `summands` function to handle factors instead of numbers.

This modification should enable you to handle expressions like  $42+23*36+42/42-1$ .

### What's next?

In order to finish this project, you need to work with a few more things, such as: parentheses, whitespaces, error handling,...



Parentheses are sub-expressions. A sub-expression can be parsed in the same way as an entire expression: **another recursive call...**