



# UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO.

## Team Gremlins

Compiler

ING. Norberto Jesús Ortigoza Márquez

**Barrientos Veana Luis Mauricio.**

**González Pacheco Leonardo Alonso.**

**Martínez Matías Joan Eduardo.**

**Rosales Romero Ricardo.**

## **Progress review**

During the previous 2 deliveries, the requirements requested by the client could be correctly prepared; the complications that have arisen throughout the project have been solved thanks to teamwork; however, we still need to debug the compiler and make it more efficient; the corresponding tests that have been carried out still require validation.

Work began on the git branches to make modifications.

The documentation has been expanded so that it can be better read and understood.

Details were debugged in the code.

The labels are being continued on GitHub so that the versions made can have greater order and organization.

## **Aspects to improve**

Communication with the client must improve, as well as with team members.

Bugs should be debugged, and more testing done on the code.

For the last installment there should be a more formal presentation of the project showing the software engineering methodologies that were implemented.

Improve the use of tools for creating the project progress review

## **Activity planning**

Tasks	Week 1	Week 2	Week 3	Week 4
Evaluation of goals agreed in the previous installment				
Read documentation from Nora Sandler.				
Updates in lexer and parser				
Implementation of the grammar to modify the tree and its path in the creation of the order of operators				
Debug the code and upload changes to GIT and GIT HUB.				

## Subject

### Binary Operators

Adding several binary operations (operators that take two values):

- Addition +
- Subtraction -
- Multiplication \*
- Division /

As usual, we'll update each stage of the compiler to support these operations.

### **Important concepts that were applied.**

#### Lexing

Each of the operators above will require a new token, except for subtraction. It gets tokenized the same way whether it's a subtraction or negation operator; we'll figure out how to interpret it during the parsing stage. Arithmetic expressions can also contain parentheses, but we already have tokens for those too, so we don't need to change our lexer at all to handle them.

Here's the full list of tokens we need to support. Tokens from previous weeks are at the top, new tokens are bolded at the bottom:

- Open brace {
- Close brace }
- Open parenthesis (
- Close parenthesis )
- Semicolon ;
- Int keyword int
- Return keyword return
- Identifier [a-zA-Z]\w\*
- Integer literal [0-9]+
- Minus -
- Bitwise complement ~
- Logical negation !
- Addition +
- Multiplication \*
- Division /

## Parser

It was important to understand the way of creating the tree including the operations previously seen because the problem that arose with the grammar previously was that it does not handle the precedence of the operator.

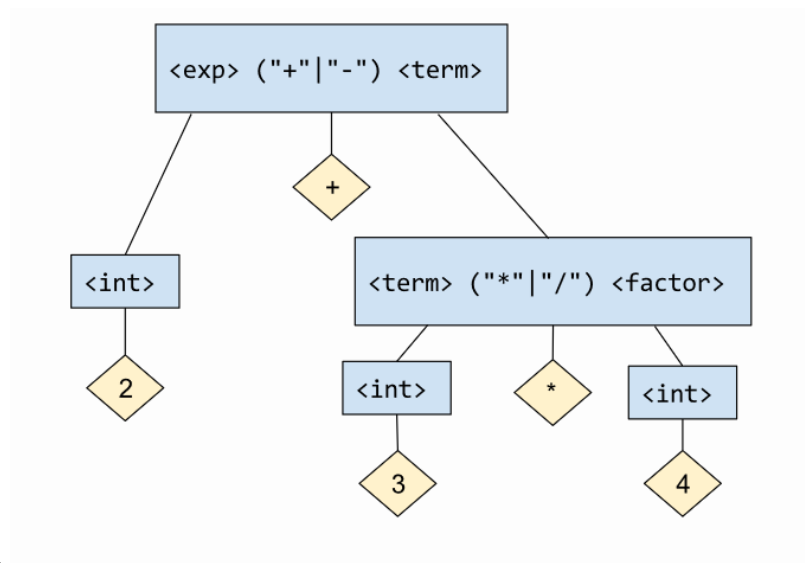
The problems detected were:

With unary operators: they always have a higher priority than binary operators.

A unary operator should only be applied to a whole expression if:

- the expression is a single integer (e.g.  $\sim 4$ )
- the expression is wrapped in parentheses (e.g.  $\sim(1+1)$ ), or
- the expression is itself a unary operation (e.g.  $\sim!8$ ,  $\sim\sim(2+2)$ ).

To express this, we're going to need another symbol in our grammar to refer to "an expression a unary operator can be applied to". We'll call it a factor. We'll rewrite our grammar like this:



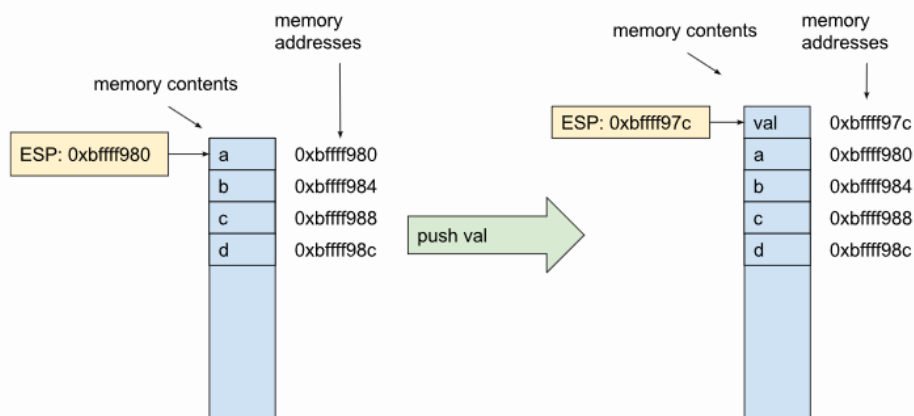
To solve each of the problems, we based ourselves on the documentation by Nora Sandler, where the complexity and solution of the operators that will form the tree are described in more detail.

## Code Generation

The code generation had its complications in the parser, but we were able to solve the grammar and the compiler correctly generated the operations to be performed. To handle a binary expression, like  $e1 + e2$ , our generated assembly needs:

- Calculate  $e1$  and save it somewhere.
- Calculate  $e2$ .
- Add  $e1$  to  $e2$  and store the result in EAX.

The challenge in this part was understanding that we needed a place to store the first operand. Saving it in a registry would be difficult; the second operand may contain subexpressions, so you may also need to save intermediate results to a register, possibly overwriting  $e15$ . Instead, we will save the first operand to the stack.



## Learning Obtained

We have realized that following a methodology when programming makes our interaction as a team more efficient, so for the last installment we will establish a stricter plan to improve all those details that are missing in the compiler. The tests are important because with them we verify that the previous deliveries are correctly elaborated, and the modularization of the code is a broad advantage since errors can be easily detected.