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.

Index

Introduction

Client Norberto Jesus Ortigoza Marquez requires a Commands that compile C language codes. The codes that need to be compiled. Provided by the same client with the purpose of all of them being compiled correctly generating the corresponding executables.

Norberto also requests that the program be created under other specific which are detailed in the requirements. Also ask for an installation manual and use so that the program can be used by any user with basic command-line knowledge. Finally, the client offers a personal repository on GitHub in order to give monitoring the project and its generated versions. Also, making a calendar of deliveries of project progress.

When making deliveries, Norberto will request explain the architecture of the program and the functionality of the source code also show a suite of tests attached to the codes provided by the client

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 GitHub 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.

Lexing

A grammar is the set of rules that define the words that a language recognizes, accepts and / or generates from a given alphabet. It is the first phase of the compiler and is known by the name of Scanner. It transforms a set of characters that are read from the source program one by one as input and output lexical components or also known as tokens, which will then be used by the parser. And it is responsible for recognizing identifiers, keywords, constants, operators, etc.

The lexical parser works at the request of the parser by giving it a lexical component every time the parser needs it. The lexical components are specified using regular expressions.

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

The second phase of the compiler is known as parsing or parsing. The parser uses the tokens produced by the lexical parser to create a tree structure that represents their grammatical structure.

This analysis is necessary to determine if a series of tokens supplied by the lexical analysis are valid in a certain language, that is, if the sentence has the correct structure or form. But not all syntactically correct sentences are valid, semantic analysis will be needed to determine this accurately.

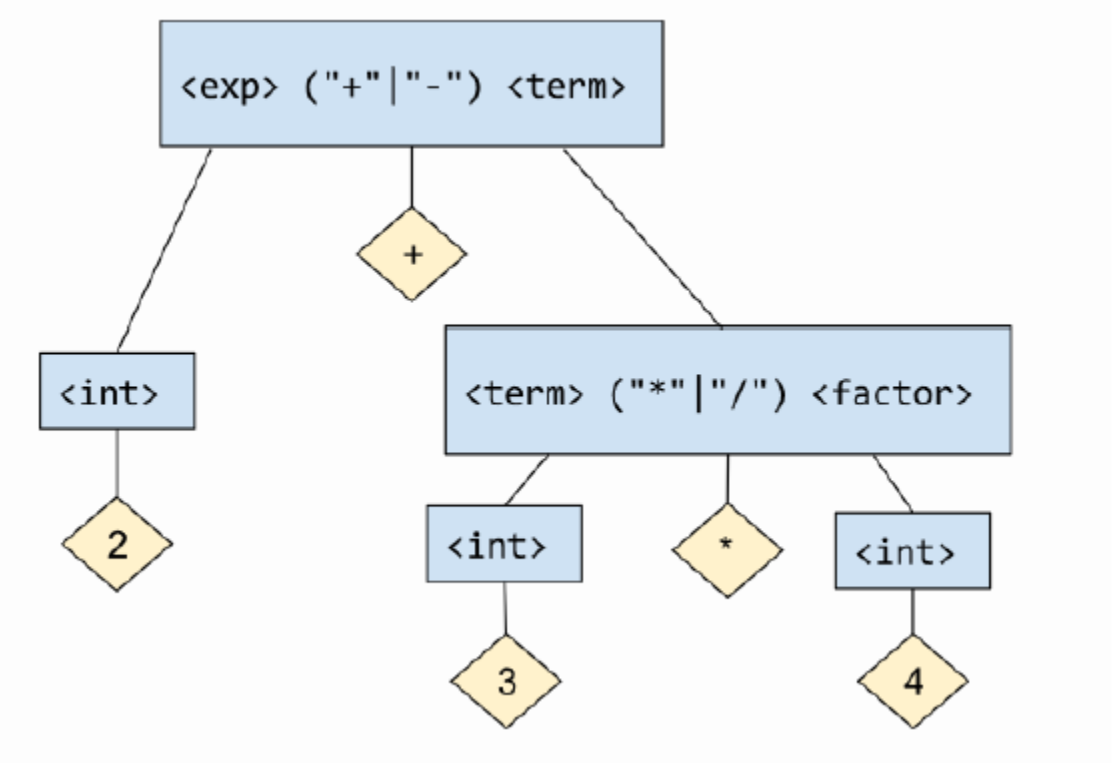
A parser takes the output of the lexical parser in the form of a string of tokens and parses it using the production rules to detect errors in the code. The output of this is a syntax tree. In this way, parsing performs two tasks, grammatically analyzing the code, finding errors, and generating a syntax tree as output.

The parser must analyze all the code even if there are errors, this is why error recovery strategies are used

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. ~4)
- the expression is wrapped in parentheses (e.g. ~(1+1)), or
- the expression is itself a unary operation (e.g. ~!8, -~(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

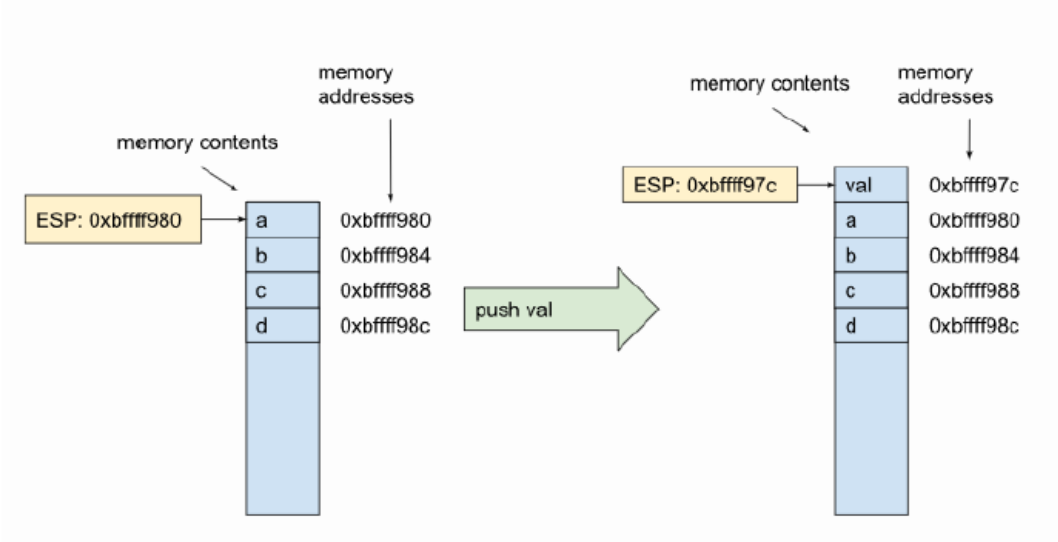
The code generator is the phase through which a syntactically correct program is converted into a series of instructions to be interpreted by a machine. The input for this phase is represented by a syntax tree, the memory locations are selected for each of the variables used by the program. Then each of the intermediate instructions is translated into a sequence of machine instructions that executes the same task. In such a way that it generates assembly code.

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 it 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.

References

- https://norasandler.com/2017/12/28/Write-a-Compiler-4.html
- http://blog.pucp.edu.pe/blog/tito/2019/01/05/crea-tu-propio-compilador-casero-parte-1/
- http://blog.pucp.edu.pe/blog/tito/2019/01/13/crea-tu-propio-compilador-parte-4-creando-un-analizador-lexico/
- http://blog.pucp.edu.pe/blog/tito/2019/02/09/crea-tu-propio-compilador-parte-6-analisis-sintactico/