C--

Programming Languages 19-20

Alex Almansa Marcel
Llopis Marc Llort Paula
Rodríguez Oleksiy
Byelyayev

index

Introduction

The goal of this project is to develop a programming language along with a compiler that can pass the high-level code to the MIPS language.

Our language is called C--, because the syntax we have chosen is very similar to that already existing in languages such as C or C ++.

Because the number of hours that can be devoted to the project is limited, the functionalities of the language are very limited, as we only have one type (int), basic operations (sums, subtractions), logical operators (<,> and ==) along with if and while.

```
int num = 0 ; int a = 0 ; int b = 1 ;
int result = 10 ;



while ( 1 < 2 ) {num = a + b ;

        a = b ;
        b = num ; if (result <num) {

                num = a + b ;
        }}
```

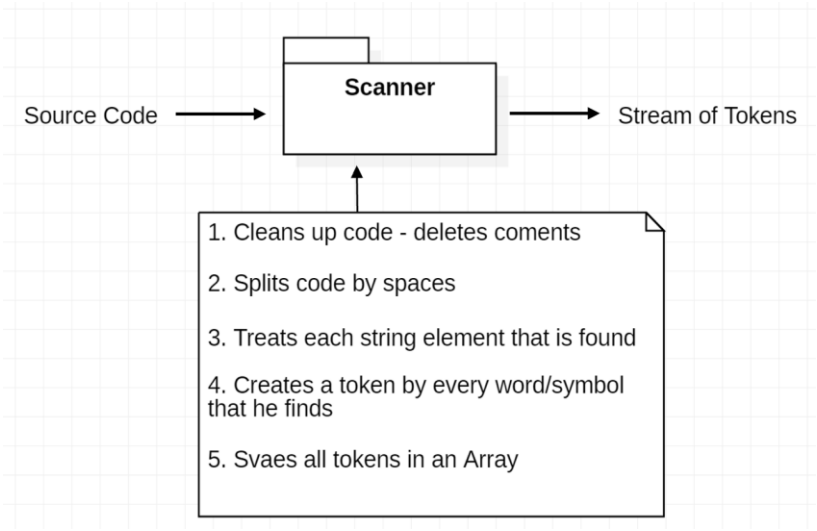
It is for this reason that it has seemed appropriate to us to call the language C--, as it could be seen as a rather cropped version of the original language.

Compiler modules

Scanner

The first module implemented in this project was the Scanner. This module receives the code that has been entered and prepares it to be able to process it, that is to say, to clean it. The first thing this module does is remove, if it finds, comments in the code, as these are not useful for the compiler. It also skips blank spaces, tabs, and line breaks. It recognizes them, but does not generate any tokens.

This module also counts the number of lines. This serves to give the possible errors that can be given with the corresponding line number.

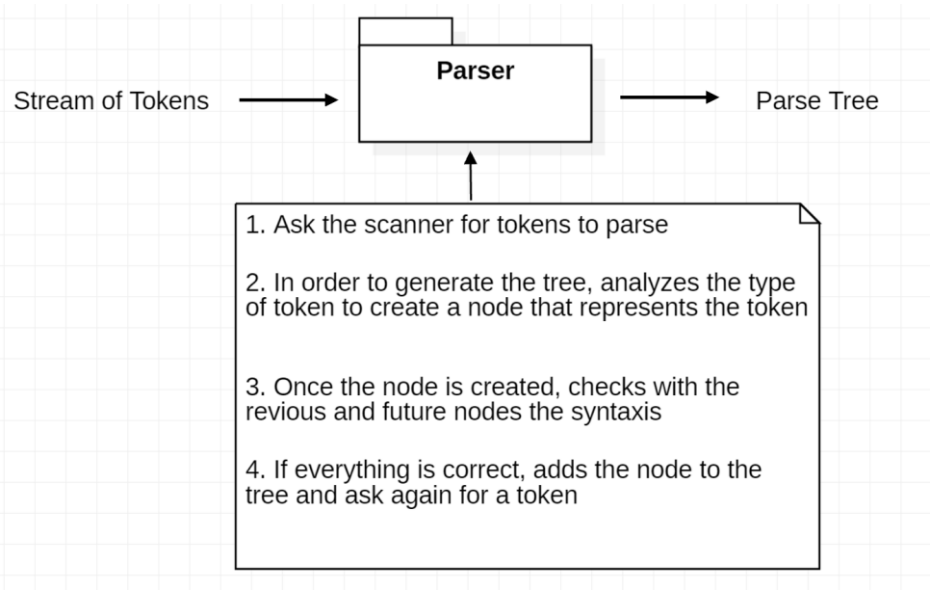


Parser

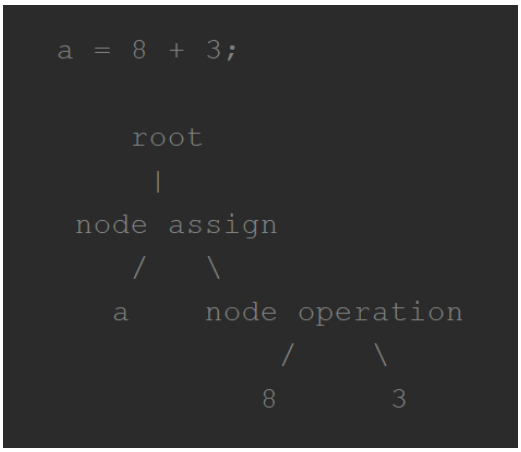
Once the code parsing module is ready, a code parsing is done. The Parser takes the tokens generated by the scanner as input, and generates a tree. At this stage, it is verified that the expression of the tokens is semantically correct.

This module is used for further data processing.

As it requested tokens in the previous module, the Scanner, looks for the correct syntactic derivation from the initial token, and generates the tree from top to bottom.



First of all it generates the node *root,* the father. Depending on the tokens you are requesting, it generates a type of node. Depending on the type of node it generates, in order to check that syntactically the code is correct, it expects the next token to be of a specific type. An example of the tree that is generated would be the following:



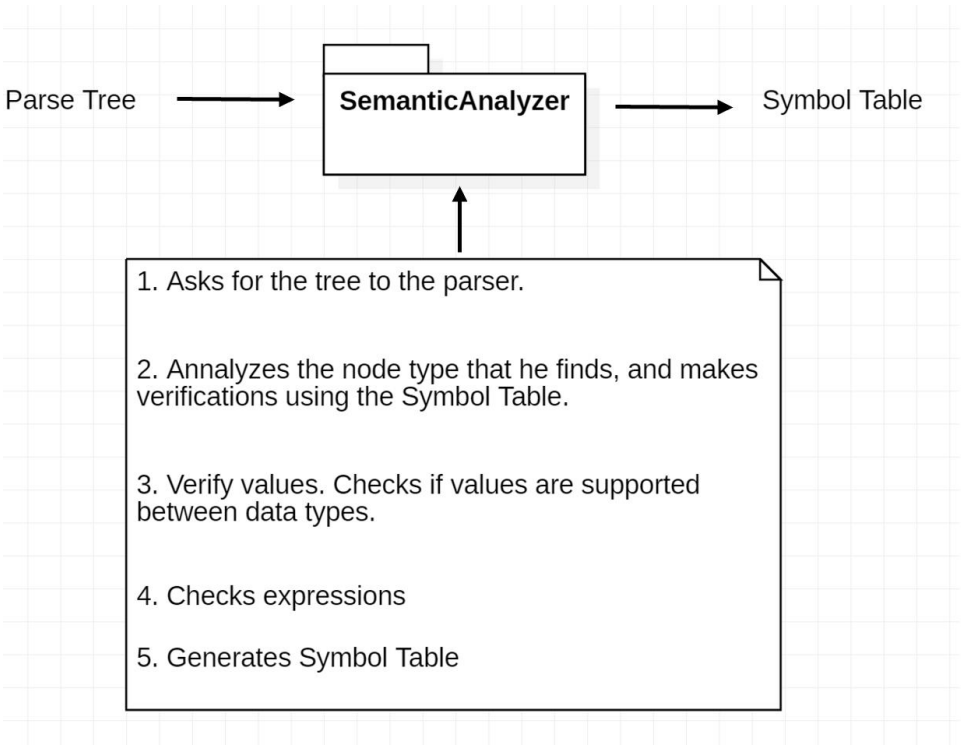
Semantic

The next step, once the tree has already been generated, comes the Semantic module. The semantic parser is the phase of the compiler that verifies the definitions of the identifiers, that is, it checks whether the analysis of the constructed tree follows the rules of the language. Verify that each expression has a correct type and that the use of this identifier is consistent with its statement.

All information about the identifiers is saved in the Symbol Table.

Verify that the semantics of the tree are correct as follows:

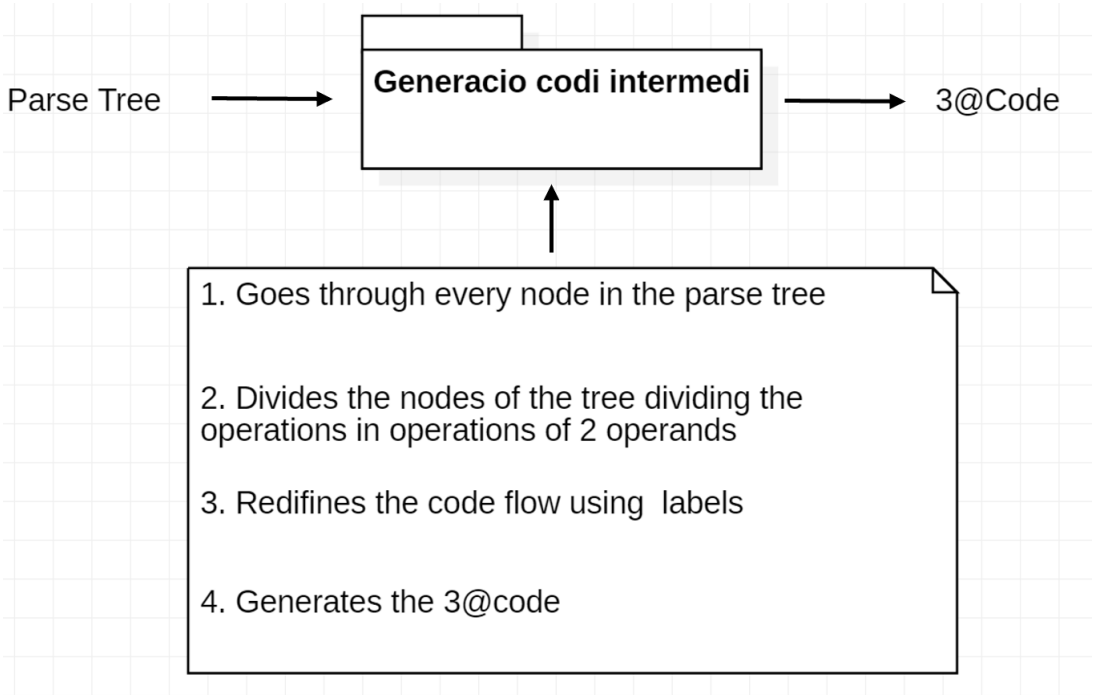
- Verification in the declaration of a variable, seeing that this one has not been declared previously. If you see that it is not in the symbol table, insert it.
- Verification that the use of this identifier is consistent with the statement, ie do not accept *int num = 'a'*.
- Verification of the conditional operations of the loops and the conditions, for example, that these are of the same type.



Intermediate code generation

Once the semantics module has verified the tree generated by the parser, it results in the generation of the intermediate code. The intermediate code generator divides an expression into subexpressions. Generates the call *three-address code,* which represents these subexpressions in four-way code, where the fields are the operator, the first argument, the second argument, and the result.

As you generated these subexpressions, a table with all these expressions was generated, and you redefined the execution flow using tags and *GOTO.*

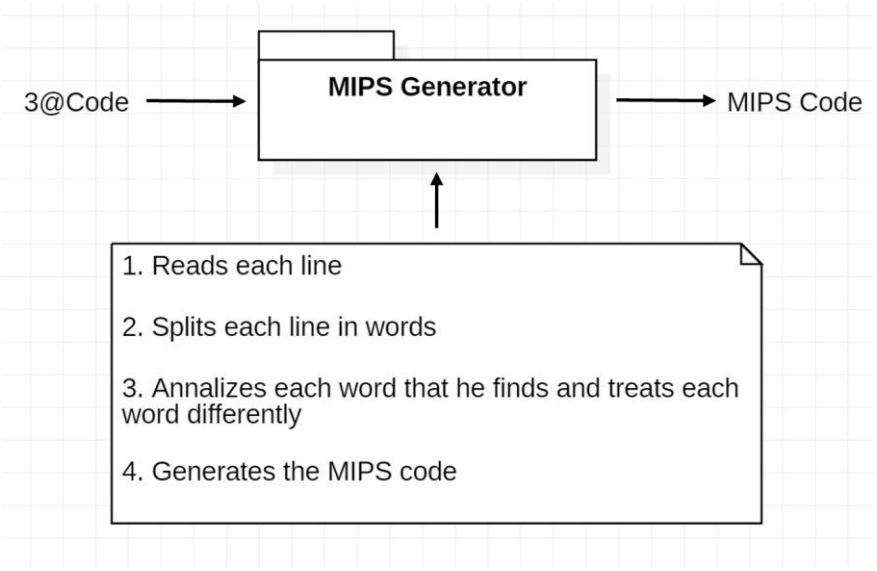


MIPS Generator

This module receives the symbol table and the code generated by the previous module, in order to generate the MIPS code.

It traverses each line of the code it receives, and from each line it traverses each word that the line contains. Treat each word differently, depending on the type of word. That is, it traverses each word in each line, and depending on the type of word, which can be operation, a variable, a process (if or while), or an operator.

Each record of the architecture is defined by a $, and this code is divided into two parts. The first part of the code is reserved for the variable declaration and code section.



An example of what the MIPS code would look like would be:

```
int num = 0;
int a = 3;

if (a < 4) {
    num = num + 1;
}
```

This is the code entered.

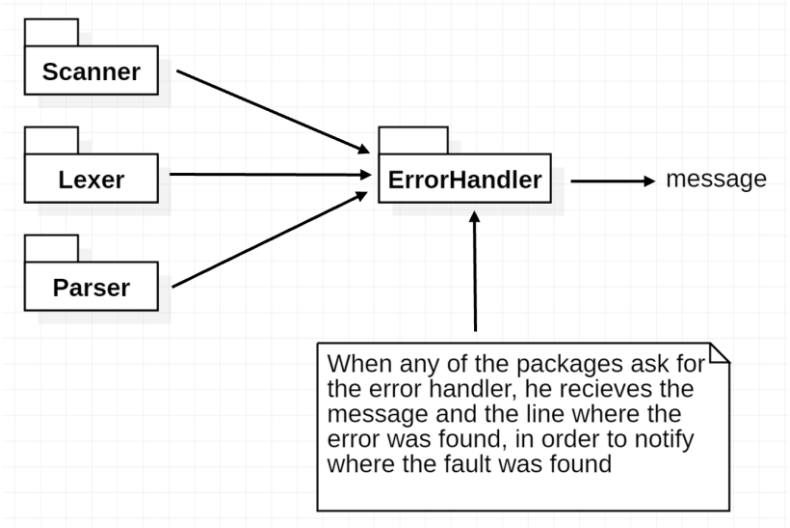
This would be the output, where we can see that at the beginning of everything we have the declarations of the variables, and then the code section.

```
IF a < 4 GOTO L0
 GOTO L1
L0
t0 = num + 1
num = t0
L1

MOVE $fp, $sp
SUB $sp, $sp , -16

LW    $num, 0
SW    $num, -12($fp)

LW    $a, 3
SW    $a, -4($fp)
```

```
LW    $a, 3
SW    $a, -4($fp)

LW     $if, -4($fp)
SLT    $if, 4, $if

BEQ    $if, $zero, L0


BNE    $if, $zero, L1

L0:

LW    $t0, -12($fp)
ADDIU    $t0, $num, 1

SW    $t0, -12($fp)
```

Error Handler

This module is used by the Scanner, Lexer and Parser modules. This module is used to warn, at runtime, of code errors that it encounters as the code is analyzed.



At the Tokens Analyzer level, it alerts you if there are any words or symbols in the code you are analyzing that you cannot find in the dictionary.

At the parser level, while it is generating the tree, it will warn if:

- I was waiting for one type of Token and found another
- If any symbol is missing, such as a (, {, = or a;
- If the fixed structure of the IF and WHILE is not correct
- …

At the Semantic level, it will warn if:

- One variable has not been previously declared.
- If the type to which the match was made is incorrect
- If the IF or WHILE condition is incorrect
- …

This will display an error message giving the line where it found it, and gives an answer of what may be happening.

We propose an example to see how this module works.

```
int num = 3;
int a = b;
```

```
> Root > node_declaration > leaf > [int]
> Root > node_declaration > leaf > [a]
Error: VARIABLE [b] IS NOT DECLARED OR IS OF INCOMPATIBLE TYPE
```

Grammar

Functions

<••••••••> :: = •••• () {<••••••••••>}

<••••••••••> :: = <••••••> | <•••••••••••••> | <•••••••••••••>

<••••••> :: = <•*h*•••••••••••••>

<•••••••••••••> :: = <••••••••••••>

<•••••••••••••> :: = <•••••••••••••••> | <•*h*•••••••••••••> |
<••••••••••••••••••>

Assignment

<•••••••••••••••> :: = <••••••••> *"="* < •••••••••••••••>

<••••••••> :: = <••••••••••>

<••••••••••> :: = <•••••••>

<•*h*•••••••>:: = <•••••••> | <•••••> | <•••••••>

<•••••••>:: = "•" | "•" | "•" | " *Ç "/"*• "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|

"" " | "•" | "•" | *"Ç" / "*• "|" • "|" • "|

"• "|" *h* "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • "|" • " | "•" | "•" | "•" | "•" | "•" | "•"

<•••••>:: = " *0 "/" 1 "/" 2 "/" 3 "/" 4 "/" 5 "/" 6 "/" 7 "/" 8 "/" 9 "*

<•••••••>:: = "|" | "" | "!" | "#" | "$" | "%" "&" | "(" | ")" | "*" | " + "|", "|"
- "|". "|" / "|": "|"; "|

">" / "=" / "==" / "<" / "?" / "@" / "[" / "\" / "]" / "^" / "_" / "` "/ "{" / "}" / "~"

<•••••••>:: = *0 / 1 / 2 / 3 / 4 / 5 / 6 / 7 / 8 / 9*

Type

You are: To represent an integer.

**<•••••••••••••••>:: = *"••• "*< •••••••>

Arithmetic operations

Sum (+): Int, Float

<••••••••••••••>:: = <••• 1> "+" < ••• 2>

Rest (-): You, Float

<•••••••••••••••>:: = <••• 1> "- "< ••• 2>

Comparison operations

<••••••••••••••••••> :: = <•••••••••••••••> == <•••••••••••••••> |

<•••••••••••••••>! = <•••••••••••••••> | <•••••••••••••••> <
<•••••••••••••••> |

<•••••••••••••••>> <•••••••••••••••>

<•••••••••••••••>:: = <•••••••••••••••> | <•••••••••••••••> | <•h•••••••
> | <•••>

Conditional

<•••••••••••> :: = •• (<••••••••••••••••••) {<••••••••••>}

Loops

<•h•••••••••••••> :: = •h••• (<••••••••••••••••••>) {<••••••••••>}

Language Specification

Tokens are basic elements of a programming language because, if we look at it from the point of view of syntax, they are the terminals of grammar. Keywords, identifiers, operators, and constants are examples of tokens.

As we mentioned before tokens are formed during the lexical analyzer, also known as *Scanner.* Here we show the data corresponding to our class.

Once this little introduction is done, all the words reserved in the compiler will be displayed.

Symbol	Type	Lexema	Definition
main	0	main	Used to define the main function
int	1	int	Enter
endOfFile	404	eof	Defines the end of the code
number	11	[0-9] +	Defines a numeric value
identify	10	[a-zA-Z0-9] +	It is used for variable names
conditional	101	if	Conditional (true / false)
loops	102	while	Loop based on Boolean condition
undef	400	undef	It is used to define variables unknown

As for the operators and the special symbols, each of them also represents a token, these are the ones implemented in the compiler.

Symbol	Type	Lexema	Definition
sum	20	\\ +	Sum operation
rest	22	-	Operation remains
assignment	21	=	To assign a value
Semi-colon	200	;	To mark the end of the line
Open parentheses	204	\\ (	To prioritize an operation
Close parenthesis	205	\\)	To prioritize an operation
Open bracket	206	\\ {	Mark the beginning of the *statement* in if or while
Close bracket	207	\\}	Mark the end of the *statement* in if or while
equality	100	==	To find out if two variables are equal
Not the same	102	! =	To find out if two variables are different
Bigger than	103	>	To find out if a variable is greater than one other
Smaller than	104	<	To find out if a variable is smaller than one other

Phase work methodology

Sprint 1 & 2 (02/03 - 14/03)

The first two Sprints served as an introduction to the big project we had ahead of us. During these first two weeks the epic incidence was that of Definition of the programming language, that is to say that it was a question of looking for information to begin to become familiar with terms like grammar bnf, dictionary (reserved words), tokens or control flow. Apart from the first knowledge we could get the idea was to start setting goals for our compiler.

We quite agreed that we wanted to make a C-based language, considerably reduced as is obvious. It should be emphasized that at that time we set ourselves goals that were perhaps too high, not out of respect for time but because of the joint work of all the subjects.

However, these two sprints served indirectly to advance the memory because thanks to the formal documentation there were sections that we already had done. Once this document was finished, it was time to start coding. That is why from that moment on the epic impact was that of Design and implementation of the compiler. It was also from this time that confinement in the country began

Sprint 3 & 4 (14/03 - 26/03)

When it came to coding we all agreed to use the Sourcetree program as Bitbucket itself recommended it. With these two tools together with the Jira we already had everything necessary to avoid version conflicts, or who makes this particular incident, who reviews etc.

These sprints represented the lexical analyzer and the semantic analyzer of the compiler. This means that a first implementation of the Scanner and the Parser was carried out together with the classes Token, Keyword, Constants and everything necessary for the Nodes tree. Each of these classes are explained in depth later. In these two phases we started to see that being 5 in the group there was not much difficulty in reaching the goals set week by week. Yet it is also true that we did what was right and necessary to achieve it.

At first it seems that the whole issue of incidents, tasks, branches and sub-branches seems difficult but with a week it was clear that it did not require too much effort. On the other hand, what was clearly seen was how useful it could be, as the idea of always having a functional version meant that it could always be tested without any problems.

Sprint 5 (26/03 - 02/04)

This Sprint was mainly used to fix all the problems arising from the first implementations as there were several things we did not do properly. For example the function traversing the node tree was redone from scratch, as was the symbol table. New comparison constants were also added to begin making a first implementation of the if conditional. To do this it was also necessary to regulate the use of parentheses and brackets. At the end of this phase we already had in operation the declaration and allocation of integers, along with arithmetic operations such as addition.

Sprint 6 (02/04 - 30/04)

This sprint was the longest of all as there was Easter in the middle, so it meant a lot of progress in the project. The first extremely important goal is that the if was fully implemented with all that it entailed. This means that we had managed to make comparisons between integers in three different ways, equal, greater than, and less than.

It was also the time when we implemented the intermediate code, also known as 3 address code. This concept introduced in class was already used to start optimizing the code as it separated complex operations into much simpler operations. To test the code, different test files were created to be able to show what was going on and also what kind of errors came out depending on what was inserted.

Last but not least, the first implementation of the MIPS code was carried out after researching this concept. It must be said that it was not too easy to understand as this type of language is not self-descriptive and required a lot of googling.

Sprint 7 (30/04 - 07/05)

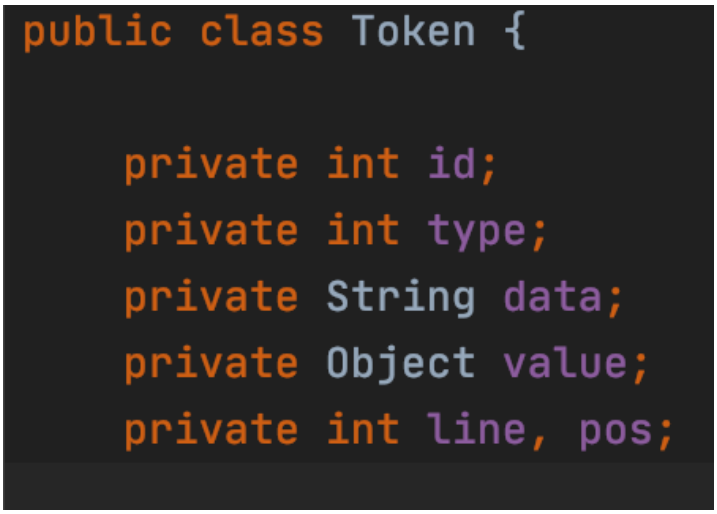
This sprint was the last of all and mainly tried to perform all the phases to implement the while loop. Obviously it also served to do a final review of the entire code and finish polishing it.

Once we finished the while it was time to start doing the memory and presentation, along with the demo to show all that our language is capable of. We considered implementing the else or the forum but in the end we discarded the idea because we did not have as much time, in addition the teacher had already told us that we had everything we needed.

Own structures

Token:

The Token class is the most basic structure of our compiler and allows us to store information regarding each of the possible combinations of characters (words) written in text files following the grammar established by the compiler. This class is generated in the phase of the lexical analyzer or Scanner, which makes a "split" by spaces and processes each string to detect if it matches a reserved word (keyword) and otherwise represents the name of a variable.



The “type” attribute stores information regarding the type of token in question, and may be one of those defined in the Constants class that refers to a reserved word or string, referring to the name of the token definition. some variable.

The “data” attribute stores the string or combination of characters that refer to that token in question.

Finally, the “line” and “pos” attributes refer to the line number and position in which this token is located.

Keywords:

Keywords enumeration allows us to verify using Regex whether a character combination is a reserved word of the language or not, this class / enumeration is used in the Scanner to validate text files written according to the grammar of our C-- language.

This class offers a static "checkKeyword" method that allows the Scanner to verify the input being analyzed and return a token that will represent the type of string analyzed.

```
public enum Keywords {

    MAIN(Constants.MAIN_PATTERN) {
        public Token generate(String input, int line, int pos) {
            return new Token(Constants.MAIN_TYPE, input, line, pos);
        }
    },
    INT_TYPE(Constants.INTEGER_PATTERN) {
        public Token generate(String input, int line, int pos) {
            return new Token(Constants.INTEGER_TYPE, input, line, pos);
        }
    },
    NUMBER(Constants.NUMBER_PATTERN) {
        public Token generate(String input, int line, int pos) {
            return new Token(Constants.NUMBER_TYPE, input, line, pos);
        }
    },

    UNDEF(Constants.UNDEF_PATTERN) {
        public Token generate(String input, int line, int pos) {
            return new Token(Constants.UNDEF_TYPE, input, line, pos);
        }
    },
    STRING(Constants.STRING_VALUE_PATTERN) {
        public Token generate(String input, int line, int pos) {
            return new Token(Constants.STRING_VALUE_TYPE, input, line, pos);
        }
    };

    protected Pattern pattern;

    Keywords(String pattern) { this.pattern = Pattern.compile(pattern); }

    public abstract Token generate(String input, int line, int pos);

    public static Token checkKeyword(String input, int line, int pos) {
        for (Keywords kw : Keywords.values()) {
            if (kw.pattern.matcher(input).matches()) {
                return kw.generate(input, line, pos);
            }
        }
        return UNDEF.generate(input, line, pos);
    }
}
```

** Reduced version of the full code*

TreeNode:

This class represents a node that used to generate the structure of the Parser tree or compiler parser.

```
public class TreeNode {

    private Parser.nodeType nodeType;
    private TreeNode parent;
    private ArrayList<TreeNode> children;
    private Token value;
```

At the time of generating the syntactic tree, the Parser is responsible for initializing this structure with the corresponding values.

- *nodeType:* refers to the node type in question, which may be one of the following types:

```
static public enum nodeType {
    Root, node_declaration, leaf, node_assign, node_if, node_condition,
    parenthesis_operation, operation, condition_operation, statements, node_while
}
```

- *parent:* is the reference to the parent node of the current node, if it is the Root node, this value will be null.
- *children:* List of descending nodes of the current node.
- *value:* The token corresponding to this node.

Symbol:

The Symbol class corresponds to the symbols that are generated by the semantic analyzer, they are the result of processing the variable declaration nodes coming from the tree generated by the Parser, these nodes relate a token with a “type” value accepted by the compiler as a type of variable, and a token with a value “type” corresponding to a “string” and which refers to the name of the variable of the type declared above.

```java
public class Symbol {

    private int type = -1;
    private String name;
    private String value;
    private int scope = -1;
```

SymbolTable:

Class corresponding to the table of symbols of the compiler, is in charge to store the symbols that it generates in the phase of the semantic analyzer in a HashMap according to the attribute *scope* of the symbol to be inserted or queried.

When you want to insert a symbol, check if it does not previously exist in the symbol table of your scope.

```java
public class SymbolTable {

    public static HashMap<String, Symbol> scopeGlobal = new HashMap<~>();
    public static HashMap<String, Symbol> scopeMain = new HashMap<~>();

    public static boolean insertSymbol(Symbol symbol, int scope) {
        boolean state = false;

        if (getSymbol(symbol.getName(), scope) == null) {
            switch (scope) {
                case Constants.SCOPE_GLOBAL:
                    scopeGlobal.put(symbol.getName(), symbol);
                    break;
                case Constants.SCOPE_MAIN:
                    scopeMain.put(symbol.getName(), symbol);
                    break;
                default:
                    break;
            }
            state = true;
        }

        return state;
    }

    public static Symbol getSymbol(String key, int scope) {
        Symbol symbol = null;
        switch (scope) {
            case Constants.SCOPE_GLOBAL:
                symbol = scopeGlobal.get(key);
                break;
            case Constants.SCOPE_MAIN:
                symbol = scopeMain.get(key);
            default:
                break;
        }
        return symbol;
    }
```

Operand:

Very secular data structure that serves to store the information saved to generate the three address code. It is simply a class that has the name of the operand and the type of operand that will be stored.

```
public class Operand {

    private String name;
    private int type;
```

IntermediateTableLine:

This is a data structure based on the data structure explained above. In this structure you can save all the operations that our programming language can perform separately in two operands, the operation and the result.

Although ultimately this table has not been used to generate the MIPs, it is still useful for storing all the information generated by the intermediate code generator and having the information well structured.

```
public class IntermediateTableLine {

    private Operand operand1;
    private Operand operand2;

    private Operand result;

    private Operand operation;
```

Conclusions

This practice is a great way to apply the knowledge learned in class.

The methodology used, it seems to us that gradually the most important concepts of the compiler are assimilated quickly, and also the main problems appear at the same time as designing a compiler.

These problems are often related to the fact that the previous structuring is incorrect, or that there was no structuring. This fact has made us realize the great importance of designing before programming, as it saves a lot of time, and helps build a solid foundation of the compiler, which in the long run ends up avoiding future problems at other stages.

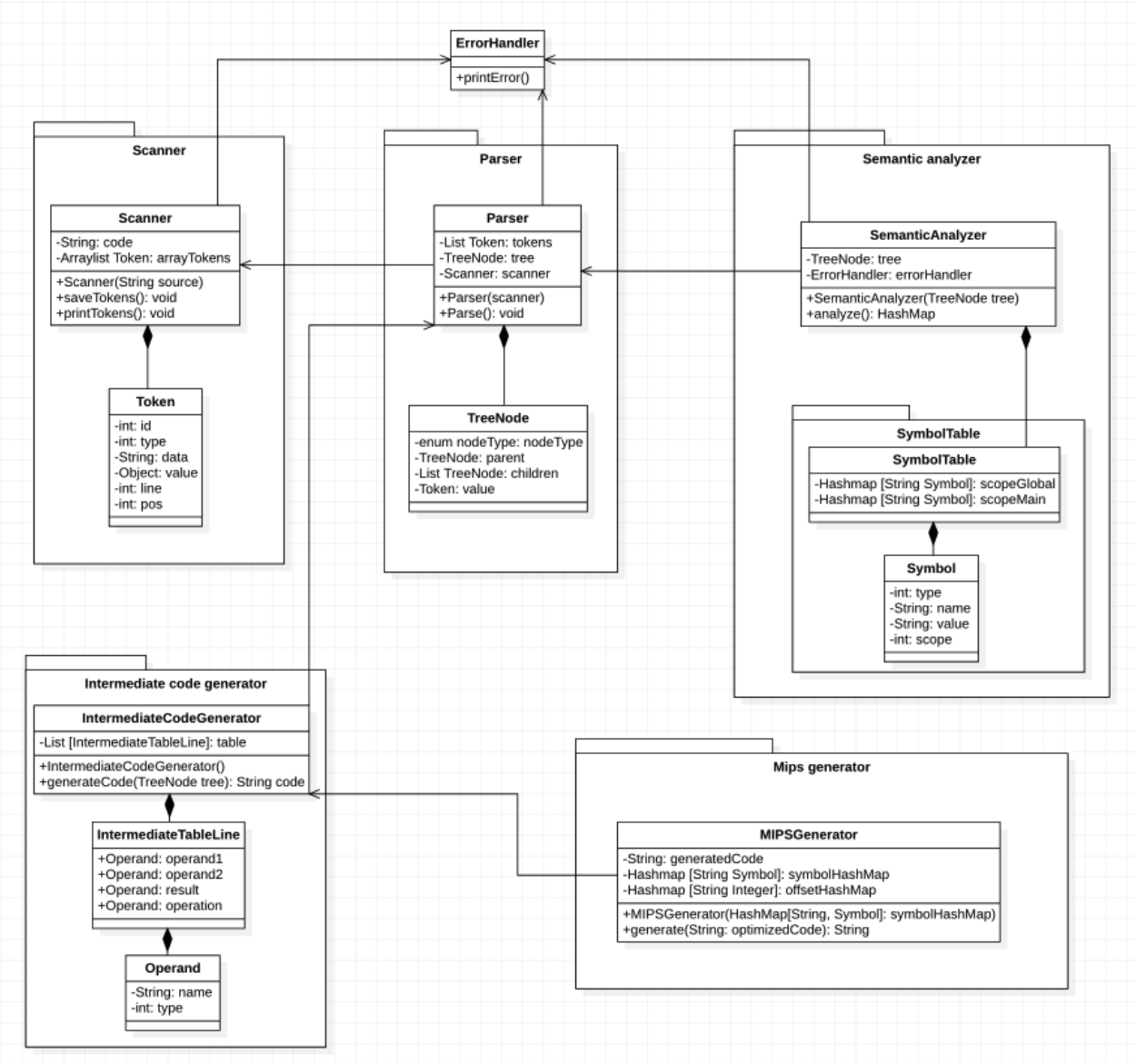
On the other hand, we found it very interesting to use Bitbucket together with Jira, since so far in most university projects, we had made maximum use of GitHub.

Classifying the tasks on the Jira board allowed us to see what problems needed to be solved, who did each task ... Having this platform also allowed us to see who had made any changes, and thus be able to ask questions. him in case of doubt.

We believe that it is important to try to bring these platforms and methodologies closer to more practical ones, because they are used in most companies.

As a final point, add that in general these types of practices that almost end up replacing classes, usually cause the knowledge learned to end up in a better assimilated way, and therefore not forget them so easily.

Class diagram



Bibliography

Aditya_04Check out this Author's contributed articles., Aditya_04, & Check out this Author's contributed articles. (2018, July 9). Compile construction tools. Retrieved from https://www.geeksforgeeks.org/compiler-construction-tools/

Ankit87Check out this Author's contributed articles., Ankit87, & Check out this Author's contributed articles. (2019, September 11). Three address code in Compiler. Retrieved from https://www.geeksforgeeks.org/three-address-code-compiler/

Compiler Design - Parser. (nd). Retrieved from https://www.tutorialspoint.com/compiler_design/compiler_design_parser.htm

Compiler Design Tutorial. (nd). Retrieved from https://www.tutorialspoint.com/compiler_design/index.htm

pp_pankajCheck out this Author's contributed articles., pp_pankaj, & Check out this Author's contributed articles. (2019, July 26). Types of Parsers in Compiler Design. Retrieved from https://www.geeksforgeeks.org/types-of-parsers-in-compiler-design/

Rajesh_Kr_JhaCheck out this Author's contributed articles., Rajesh_Kr_Jha, & Check out this Author's contributed articles. (2019, November 21). Introduction of Compiler Design. Retrieved from https://www.geeksforgeeks.org/introduction-of-compiler-design /

Compilation theory: Lexical and syntactic. (nd).

Compilation Theory: Semantics and Code Generation. (nd).

Using SPIM with Modern Compiler Implementation. (nd). Retrieved from https://www.cs.princeton.edu/~appel/modern/spim/