Universidad Nacional Autónoma De

México. Facultad de Ingeniería.

Compilers

Parse

Alumno

318218814

Grupo:

#5

Semestre:

2024-II

México,CDMX. Abril 2024

Content

Introduction

The development of compilers is a fundamental camp in computing that includes the creation of tools that are capable of translate source code of high level to executable format for a machine . The process of developing of a compiler include the following phases: lexical analyzer, parse, semantic analyzer, intermediate code generation, code optimization and target code generator.

In our previous project, we focused on the phase of lexical analyzer, in which we create a lexer for the tokens classification according to a regular expression . Once the lexical analyzer phase ends, we are in the necessity of designing an element that takes those tokens and finds them the right sense that will be defined by a grammar that is in charge of generating all the valid strings for the chosen language. We know this process as parse or syntactic analyzer.

Through this project, we will explore the design and the implementation of our parser, as well as the faced challenges and the solution proposed in the development of an integral syntactic analyzer.

Theoretical framework

Before we dive into parsing, it's important to remember the compiler definition. A compiler is a program that translates source code from a programming language into a language that can be understood by the computer. Compilers have made computers more useful than they would have been because they give programmers the opportunity to use languages that have a more natural way of expressing the solution to a programming problem.

Syntactic analysis consists of determining whether a succession of symbols in a language can be formed with the rules of grammar; that is, it belongs to language. This means that the syntax of a programming language is defined by a grammar.

The grammars that a parser accepts are context-free grammars, because it is not an easy task to understand more complex grammars and build automatons that recognize the statements it accepts. [1]

It's important to remember that a context-free grammar is made up of the tuple of four elements. Let G be a context-free grammar defined by the tuple (N, T, P, S), where:

N = No terminals

T= Terminals

P= Production rules

S = Initial Axiom

Recall that a context-free grammar is ambiguous if, for the same word in the generated language, there are two or more structurally distinct derivation trees. The choice of this type of grammars is based on the fact that what interests us about syntactic analysis is that given an input program w to decide if it belongs to the language generated by the grammar that specifies the syntax of the programming language, it is also necessary to know the sequence of productions that participate in the derivation w. In other words, we are interested in knowing the structure of w, so it is essential to work with grammars that allow us to obtain a syntactic analysis from the input program, but also that we obtain them efficiently.

As mentioned above, it is useful to visualize the parser by means of a syntax tree for input. In this tree, the root represents the starting symbol of the grammar. The tricky part of parsing lies in figuring out the grammatical connection for the leaves and root. To perform the lexical analyzer there are two general strategies that can be followed to find the derivation. [1]

Top-down parsers: begin with the root and grow the parsing tree toward the leaves. At each step, a top-down parser selects a node for some nonterminal on the tree and extends it with a subtree that represents the right-hand side of a production that rewrites the non-terminal.

In this classification we can find the parse tree with late, with recursive functions and with LL(1) grammars. We will focus in the last one.

Bottom-up parsers: Begin with the leaves and grow the tree toward the root. At each step, a bottom-up parser identifies a contiguous substring of the parse tree's upper fringe that matches the right-hand side of some production; it then builds a node for the rule's lefts-hand side connects it into the tree.

Top-Down analysis of LL(1) grammars is based on a table driven. This table is structured such that its rows correspond to the non-terminals of the grammar, and its columns correspond to the terminals, including the pseudotoken EOF. Each cell within the table (which aligns with a specific terminal column and non-terminal row) either holds a production rule or is left blank. A blank cell signifies that the sequence is not recognized. [2]

An LL(1) grammar is such that you only need to look at the first symbol at the entry point to determine which rule to use. This starts from the initial axiom and proceeds with left derivations.

We will have the FIRST and FOLLOW functions in this kind of parsers. These functions play a crucial role in LL(1) parser for several reasons. Predictive parsing, a distinguishing feature of LL(1) parsers, depends on the use of FIRST and FOLLOW sets to predict the production rule to be applied, based solely on the current nonterminal and the next input token. These sets are vital in resolving conflicts by helping to choose the correct parsing path when there are multiple possibilities. For example, when there is an alternative A | B and both FIRST(A) and FIRST(B) include the same token "a," a FIRST/FIRST conflict occurs, creating uncertainty about whether to expand A or B when "a" appears in the input. Furthermore, FIRST sets help improve parsing performance, especially by quickly determining whether a nullable nonterminal can consume any input by checking if its FIRST set includes the next token. These sets also help verify the grammar's suitability for LL(1) parsing; intersecting FIRST sets of different productions of the same nonterminal or a nonterminal that can derive an empty string with a nonempty intersection of its FIRST and FOLLOW sets indicate that the grammar is not suitable for LL(1) parsing. Lastly, FIRST and FOLLOW sets play a key role in building parsing tables, which guide the parser in choosing the right production rule based on the current input symbol and the nonterminal being processed. [3]

For the implementation it was chosen to develop a parse LL(1). Which is a parse predictive and will be use right most derivation

The grammar that generate the language is defined by the following productions

S-> returnI | intF | idA | printE

A-> (M | ; | =I

B -> ; | xH

C -> numB

E-> (J

F ->idA

G-> ; | xH

H -> numB | idG

I -> -C | idG | numB | ;

J -> cadK

K->)L

L-> ;

M-> numN | idN

N->)O | ,M

O-> ; | {

Due to each non-terminal symbol "A" , exist a group of productions of the form aB where a is terminal symbol and B is a non-terminal (in some cases B does not exist) we can assure that is a free context grammar and a regular grammar.

For more convenience in the grammar, it was chosen implement a serie of tags in some tokens

For the tokens that are reserved words such as : int ,return and print there is not tag. Same case for the punctuation signs such as: ; , () {}.

For the tokens that represent constant such as: 666 o 911 are replaced by the tag "num"

The tokens that are operators (+ - * /) were replaced by the tag "x". The operator "=" is excluded from this tag. The token that represents a number with a signed "-" wasn´t replaced either.

The tokens that represent literals such as: "Hola mundo". Were replaced by the tag "cad".

Finally, in the case of tokens that represent identifiers such as : variable1, var 2. Were replaced by the tag "id"

Once that generate the grammar it is important to obtain the first and follow of the non-terminal symbols. Given the nature of grammar and the fact that we don't have productions with epsilon we can observe that the follow element for each non-terminal is the end of the line. For example, if we try to calculate the follow of A, we are placed in the production S-> idA what follows after A is the end of the line. The same is true for all non-terminals.

Calculating the first of the elements

First(S) = { return, int, id, print}

First(A) = { (= ;}

First(B) = { ; x}

First(C) = {num}

First(E) = { (}

First(F) = { id }

First(G) = { ; x }

First(H) = { id , num }

First(I) = { id, num, -}

First(J) = {cad}

First(K) = {) }

First(L) = { ; }

First(M) = { id, num}

First(N) = {) , }

First(O) = { { }

With this we should build the parsing table

	return	int	id	print	(	=	;	x	num	-	cad	)	,	{	$
S	returnI	int F	idA	printE											
A					(M	=I	;								
B							;	xH							
C									numB						
E					(J										
F			idA												

G							;	xH							
H			idG						numB						
I			idG						numB	-C					
J											cadK				
K												)L			
L							;								
M			idN						numN						
N												)O	,M		
O							;							{	

With this we can generate strings such as:

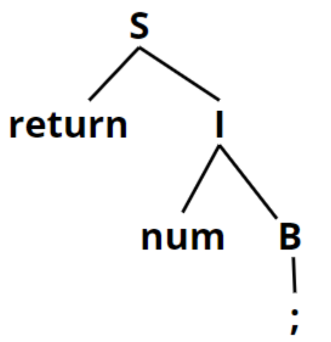
```
return -9 ;
int variable = 56*var2;
int variable2;
print(“HolaMundo”);
```

We can generate nested function

```
int main(var1){

        int funcion(variable2){

        }

}
```

It is important to mention that all instructions that aren´t functions must end with the symbol “ ; ”

Example: if we have the string return 45; it derivation tree looks this way



Development

For the code, further development was carried out in Python with the visual studio code text editor. Modifications were made to the lexer part so that it gives return+num+ type outputs; where the + symbol is merely representative that there is a space between them, this is done in the convertToken function. In the main file, the file read function is executed and then the lex function of the lexer file is called out, resulting in a list where the code with the respective token tags is shown line by line in the format return+num+;

Once you have the list of tokens, you call the parse function where you process an LL(1) parse.

Within parse we have different structures that help us to carry out the process.

Grammar: A dictionary of lists of lists, which represents grammar. Where each key (non-terminal element) has a series of lists, representing the production.

first: It is a dictionary that stores the first elements of each non-terminal parseTable: It is a dictionary that represents the parsing table, where each terminal is associated with its production, for example for terminal "x" it would look like this: x: { [B, [x , H]] [G, [x, H] }. With this structure we identify the terminal symbol, obtain the non-terminal from which it starts, and determine the non-terminal symbol to which it goes

stack: This is a list that will represent a stack in the parse process.

StackE: It is a special stack, because it analyzes line by line elements such as {} that are on different lines cause conflict, that is why this structure helps to verify the correct opening and closing of brackets.

Functions

-parse: It is the main function, in which the list of tokens of the lines of code is received, it is responsible for calling the functions that determine the first elements and the parse table. After that, we check each line of the lineTokens list, remove the "+" elements with split, and place the $ element at the end of the line and this will be known as our buffer. On the stack we place the $ character together with the non-terminal S, which is where the first production should start from. Later we call the Parse generation function, at the end we bring the buffer, a stack and an auxiliary parameter that indicates that it is the end of the line to check if there is any error.

-constructionFirst: Generate the first. We go through the productions of each of the non-terminals, given the grammar, the element 0 of each production is the first, then we save the first in a list to finally assign in a dictionary the non-terminal with its corresponding list of first.

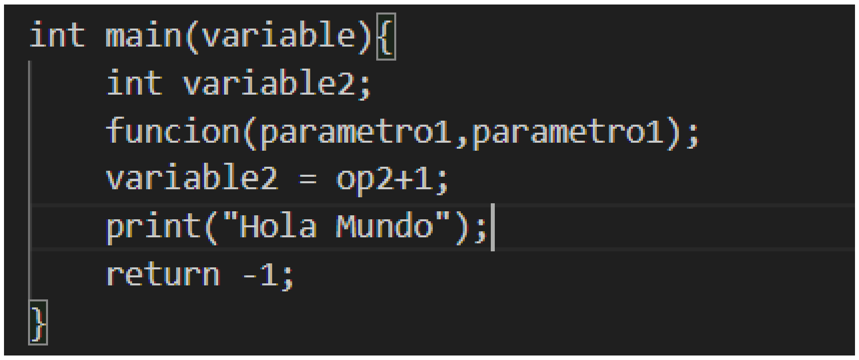
-ParseTable construction. It generates the representation of the parse table, where each terminal will be a dictionary of lists, each list will have a character at the beginning representing the non-terminal and a list that represents the generated production.

-errors: Identifies based on the main stack and the E-stack if there is an error in the code if the stack is topped by the character "$" there is no error.

For the error part, they were determined based on what is left in the stack.

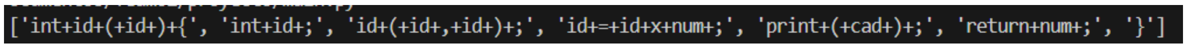
Results

If we have the following input in the document of prueba.txt



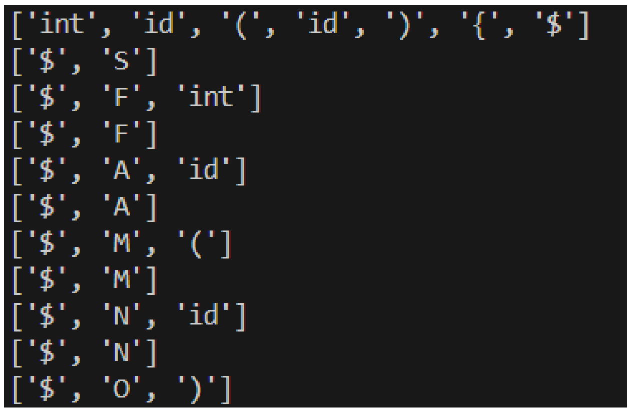
```
int main(variable){
    int variable2;
    funcion(parametro1,parametro1);
    variable2 = op2+1;
    print("Hola Mundo");
    return -1;
}
```

This is what give the lexer to the parse

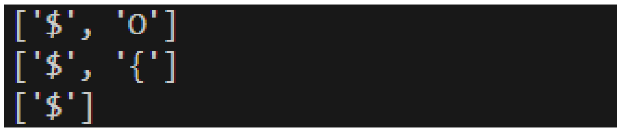


```
['int+id+(+id+)+{', 'int+id+;', 'id+(+id+,+id+)+;', 'id+=+id+x+num+;', 'print+(+cad+)+;', 'return+num+;', '}']
```

And in the parse it is read each elements in the list, showing the movements in the stack



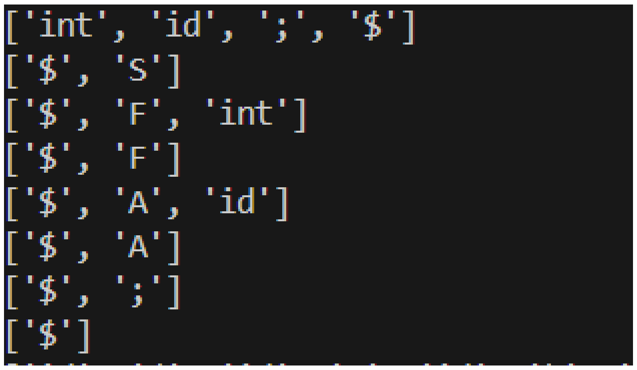
```
['int', 'id', '(', 'id', ')', '{', '$']
['$', 'S']
['$', 'F', 'int']
['$', 'F']
['$', 'A', 'id']
['$', 'A']
['$', 'M', '(']
['$', 'M']
['$', 'N', 'id']
['$', 'N']
['$', 'O', ')']
```



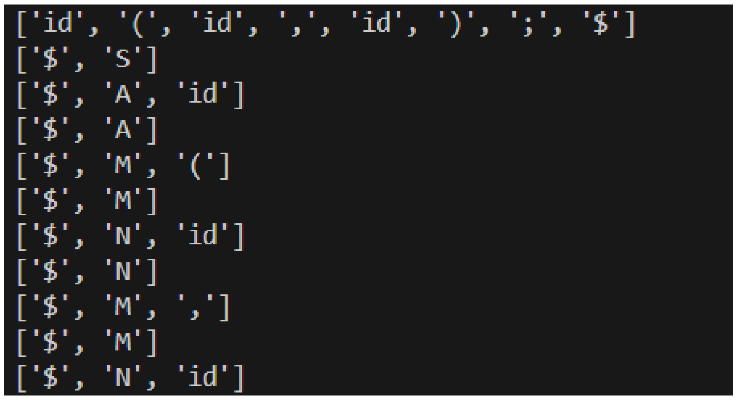
```
['$', 'O']
['$', '{']
['$']
```

At the end of every element of the list we have to have the symbol $

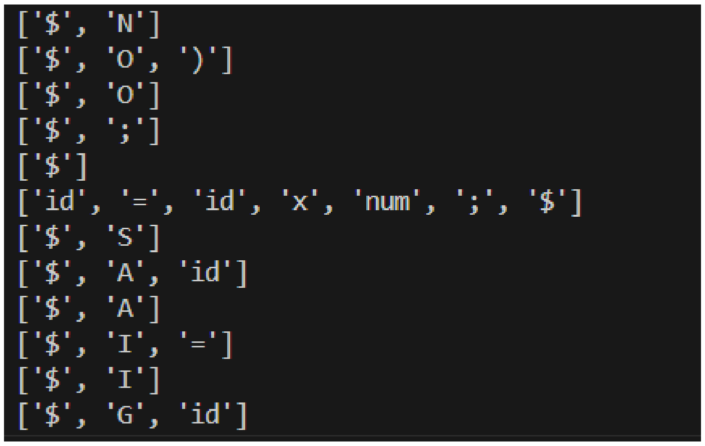
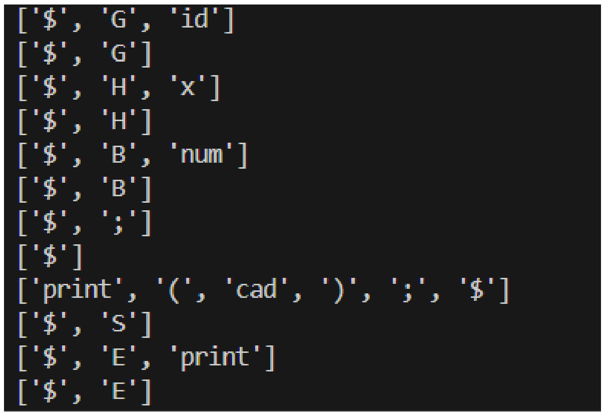
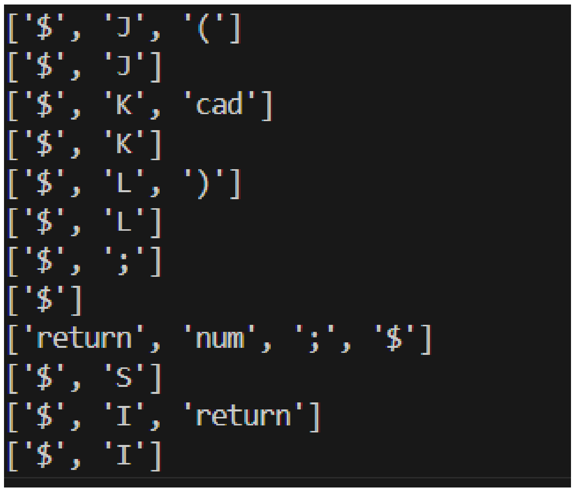
And continue with the process



```
['int', 'id', ';', '$']
['$', 'S']
['$', 'F', 'int']
['$', 'F']
['$', 'A', 'id']
['$', 'A']
['$', ';']
['$']
```

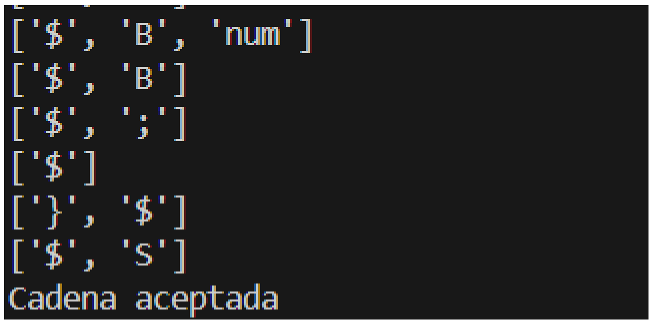


```
['id', '(', 'id', ',', 'id', ')', ';', '$']
['$', 'S']
['$', 'A', 'id']
['$', 'A']
['$', 'M', '(']
['$', 'M']
['$', 'N', 'id']
['$', 'N']
['$', 'M', ',']
['$', 'M']
['$', 'N', 'id']
```


```
['$', 'N']
['$', 'O', ')']
['$', 'O']
['$', ';']
['$']
['id', '=', 'id', 'x', 'num', ';', '$']
['$', 'S']
['$', 'A', 'id']
['$', 'A']
['$', 'I', '=']
['$', 'I']
['$', 'G', 'id']
```

```
['$', 'G', 'id']
['$', 'G']
['$', 'H', 'x']
['$', 'H']
['$', 'B', 'num']
['$', 'B']
['$', ';']
['$']
['print', '(', 'cad', ')', ';', '$']
['$', 'S']
['$', 'E', 'print']
['$', 'E']
```

```
['$', 'J', '(']
['$', 'J']
['$', 'K', 'cad']
['$', 'K']
['$', 'L', ')']
['$', 'L']
['$', ';']
['$']
['return', 'num', ';', '$']
['$', 'S']
['$', 'I', 'return']
['$', 'I']
```

At the end we show that the input code has been accepted

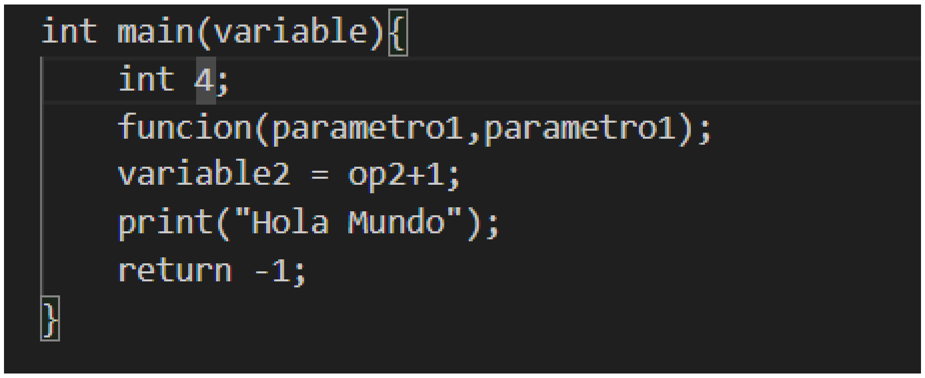
The element S in the stack at the end merge of putting {} so we support in the pilaE



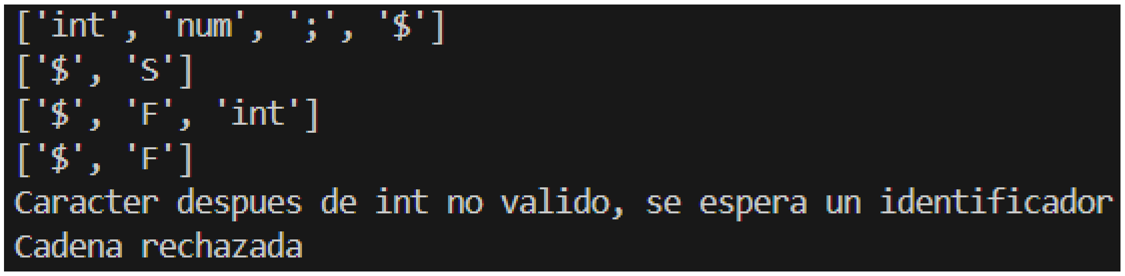
```
['$', 'B', 'num']
['$', 'B']
['$', ';']
['$']
['}', '$']
['$', 'S']
Cadena aceptada
```

In case of existing an error this one is determine because the stack have elements

If we have this input: int 4 , it is wrong for our language

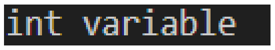


```
int main(variable){
    int 4;
    funcion(parametro1,parametro1);
    variable2 = op2+1;
    print("Hola Mundo");
    return -1;
}
```

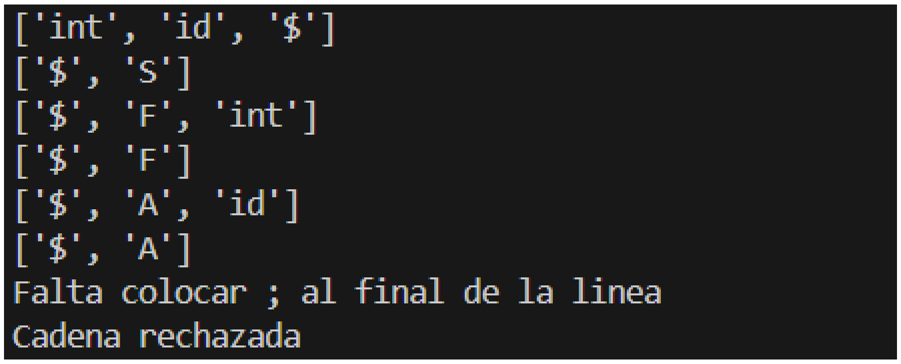


```
['int', 'num', ';', '$']
['$', 'S']
['$', 'F', 'int']
['$', 'F']
Caracter despues de int no valido, se espera un identificador
Cadena rechazada
```

If we have an instruction without ; at the end it is also mistake

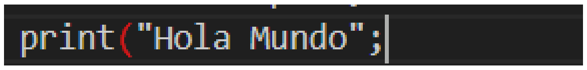


```
int variable
```

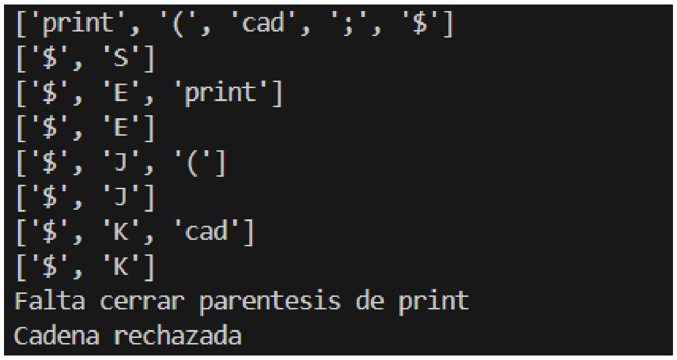


```
['int', 'id', '$']
['$', 'S']
['$', 'F', 'int']
['$', 'F']
['$', 'A', 'id']
['$', 'A']
Falta colocar ; al final de la linea
Cadena rechazada
```

If we have a print without closing (it mark a mistake

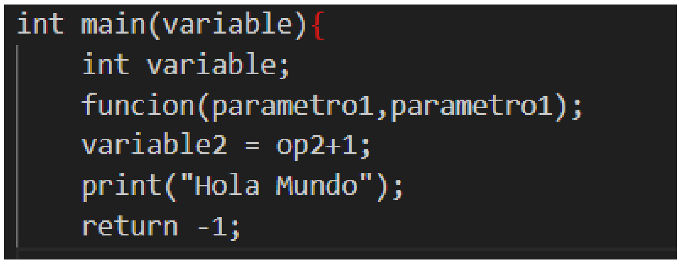


```
print("Hola Mundo";
```

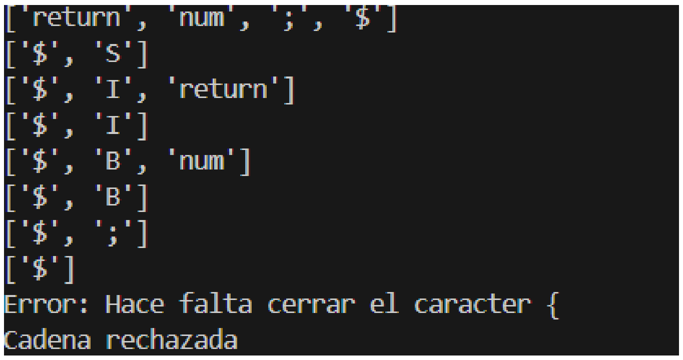


```
['print', '(', 'cad', ';', '$']
['$', 'S']
['$', 'E', 'print']
['$', 'E']
['$', 'J', '(']
['$', 'J']
['$', 'K', 'cad']
['$', 'K']
Falta cerrar parentesis de print
Cadena rechazada
```

In a function in which we have the symbol { without closing

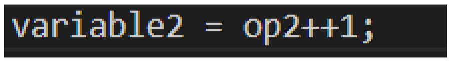


```
int main(variable){
    int variable;
    funcion(parametro1,parametro1);
    variable2 = op2+1;
    print("Hola Mundo");
    return -1;
```

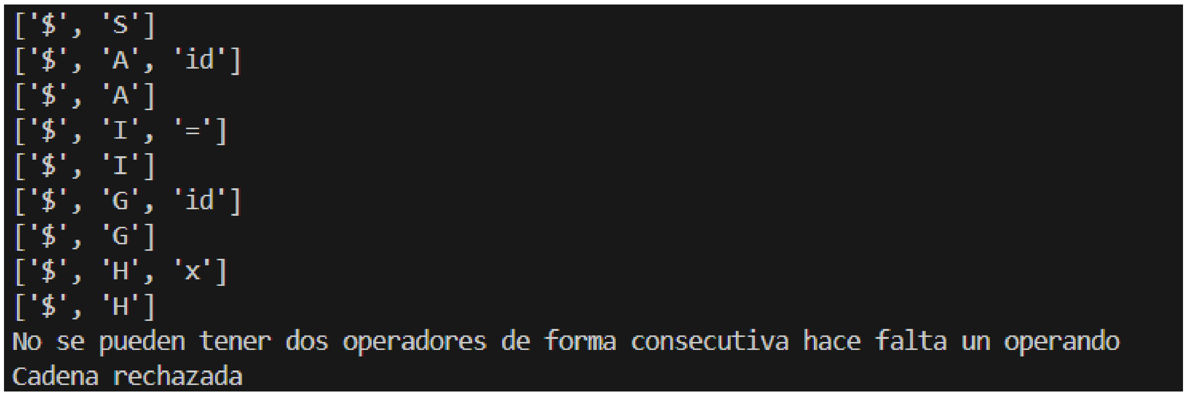


```
['return', 'num', ';', '$']
['$', 'S']
['$', 'I', 'return']
['$', 'I']
['$', 'B', 'num']
['$', 'B']
['$', ';']
['$']
Error: Hace falta cerrar el caracter {
Cadena rechazada
```

If we have two consecutive operators it also mark a mistake



```
variable2 = op2++1;
```



```
['$', 'S']
['$', 'A', 'id']
['$', 'A']
['$', 'I', '=']
['$', 'I']
['$', 'G', 'id']
['$', 'G']
['$', 'H', 'x']
['$', 'H']
No se pueden tener dos operadores de forma consecutiva hace falta un operando
Cadena rechazada
```

Conclusions

With the input strings we have been able to demonstrate that the parser implemented has demonstrated an efficient performance, accomplishing with the objective of determining that the input string can be a product of derivations of the established grammar.

It is important to mention that in the implementation of the code when calculating the first and follow , this operations are simplify by having the grammar as a regular grammar, thus there aren´t epsilon transitions making easier the construction of the dictionary that represents the parsing table. Since the implemented parse is a LL(1) in which we can elaborate a tree of analysis and the process of choosing the production is something that repeats itself constantly , we implement this process in a recursive form.

Thanks to the concepts studied in class, we have been able to implement the parse, which is a middle point between our lexical analyzer and our future semantic analyzer, in which our programming language is being defined.

Bibliography

[1] S. Gálvez Rojas y M. Á. Mora Mata, JAVA A TOPE: TRADUCTORES Y COMPILADORES CON LEX/YACC, JFLEX/CUP Y JAVACC., Málaga: Universidad de Málaga, 2005.

[2] K. Cooper y L. Torczon, Engineering a compiler, Burlington: ELSEVIER, 2011.

[3] A. F. Zúñiga Chávez , Diseño e implementación de un compilador para un subconjunto en Python, Ciudad de México: Universidad Nacional Autónoma de México, 2013.