

**PONTIFÍCIA UNIVERSIDADE CATÓLICA DO PARANÁ
ESCOLA POLITÉCNICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO**

**PEDRO HENRIQUE CAVALHIERI CONTESSOTO
JEFFERSON SOBRINHO ABBIN**

PROJETO COMPILADOR (FASE 3)

**CURITIBA
2023**

PEDRO HENRIQUE CAVALHIERI CONTESSOTO
JEFFERSON SOBRINHO ABBIN

PROJETO COMPILADOR (FASE 3)

Atividade avaliativa de Curso de
Linguagens Formais e Compiladores da
Pontifícia Universidade Católica do
Paraná

Orientador: Me. Eng. Frank Coelho de
Alcantara

CURITIBA
2023

1 FASE 1 – DEFINIÇÃO DA LINGUAGEM E ANÁLISE LÉXICA

Para definição da linguagem de programação foi definida a seguinte bloco de declarações pela lista abaixo, com declarações terminais estão em **negrito**:

```

program → block
block → {decls stmts}
decls → decls decl | e
decl → type id
type → type [num] | basic
stmts → stmts stmt | e
bool → bool || join | join
join → join && equality | equality
equality → equality == rel | equality != rel | rel
rel → expr < expr | expr <= expr | expr >= expr | expr > expr | expr
expr → expr + term | expr - term | term
term → term * unary | term / unary | unary
unary → !unary | unary | factor
factor → (bool) | num | real | true | false | string
stmt → if (bool) stmt
stmt → if (bool) stmt else stmt
stmt → while ( bool ) stmt
stmt → do stmt while (bool)
stmt → break
stmt → WritePort (string, true | false )
stmt → ReadPort (string, string)
stmt → ReadSerial (string, expr)
stmt → WriteSerial (string, string )
stmt → Writeanalog (string, expr )
stmt → Readanalog (string, string)
stmt → sleep (expr)
num → {0 - 65.536}
char → {0 - 9} || {A - z} || {A - Z} || {!@#$%`&*()_+=}
```

Para a seguinte linguagem segue 3 (Três) exemplos código com suas funcionalidades de possíveis:

EXEMPLOS

```
1) INT X;
   IF (X 12) {
       WRITE SERIAL(S3, 2301);
       BREAK;
   }
```

```
2) INT X;
   BOOL BUFFER;

   WHILE (X < 12) {
       BUFFER = !BUFFER;
       WRITE PORT(D1, BUFFER);
       SLEEP (BUFFER);
       X = X + 1;
   }
```

3)

```
4) while (x < 12){
    buffer = !buffer
    WRITE PORT(D1, BUFFER)
    SLEEP (10)
    x = x +1
}
```

```
5) X = READ ANALOG (A2, BUFFER)
   IF (BUFFER > 3000){
       WRITE SERIAL(S3, "PRESSÃO ELEVADA")
   }
```

Para a etapa da análise Léxica foi desenvolvido o código para realizar a mesmo, conforme solicitado o código é capaz de validar todos os possíveis lexemas da linguagem, usando máquinas de estado finito, implementado em Python.

Temos a primeira etapa do código onde inicializado os estados inicial

```
def __init__(self):
    self.state = 'START'
    self.lexeme = ""
    self.tokens = []
    self.position = 0
```

Abaixo esta definição dos estados de transição:

```
def transition(self, char):
    if self.state == 'START':
        if char in [' ', '\n', '\t']: # Skip whitespace
            self.position += 1
            return
        elif char.isalpha() or char == '_':
            self.state = 'IDENTIFIER'
        elif char.isdigit():
            self.state = 'NUMBER'
        elif char in ['=', '!', '<', '>', '+', '-', '*', '/']:
            self.state = 'OPERATOR'
        elif char in ['(', ')', '{', '}', '[', ']', ',', ';']:
            self.state = 'MARKERS'
        else:
            print(f"Invalid token: {char}")
            return

    self.lexeme += char
    elif self.state == 'IDENTIFIER':
        if char.isalpha() or char.isdigit() or char == '_':
            self.lexeme += char
```

```

        elif self.lexeme in ["if", "while", "break", "do", "sleep", "WritePort", "ReadPort",
"ReadSerial", "WriteSerial", "Writeanalog", "Readanalog"]:
            self.emit('KEYWORD')
            self.transition(char)
        else:
            self.emit('IDENTIFIER')
            self.transition(char) # reprocess the current character
elif self.state == 'NUMBER':
    if char.isdigit():
        self.lexeme += char
    elif char == '.':
        self.state = 'REAL'
        self.lexeme += char
    else:
        self.emit('NUMBER')
        self.transition(char) # reprocess the current character
elif self.state == 'REAL':
    if char.isdigit():
        self.lexeme += char
    else:
        self.emit('REAL')
        self.transition(char) # reprocess the current character
elif self.state == 'OPERATOR':
    self.lexeme += char
    if not self.lexeme in ['==', '!=', '<=', '>=']:
        self.lexeme = self.lexeme[:-1]
        self.emit('OPERATOR')
        self.transition(char) # reprocess the current character

elif self.state == 'MARKERS':
    self.lexeme += char
    self.lexeme = self.lexeme[:-1]
    self.emit('MARKERS')
    self.transition(char) # reprocess the current character

```

Abaixo as funções de auxiliares para definição dos tokens reconhecidos:

```
emit(self, token_type):
    self.tokens.append((token_type, self.lexeme, self.position))
    self.position += len(self.lexeme)
    self.lexeme = ""
    self.state = 'START'
```

```
def tokenize(self, code):
    for char in code:
        self.transition(char)
    if self.lexeme: # handle the last token
        self.emit(self.state)
    return self.tokensxiliares
```

```
def emit(self, token_type):
    self.tokens.append((token_type, self.lexeme, self.position))
    self.position += len(self.lexeme)
    self.lexeme = ""
    self.state = 'START'
```

```
def tokenize(self, code):
    for char in code:
        self.transition(char)
    if self.lexeme: # handle the last token
        self.emit(self.state)
    return self.tokens
```

Abaixo as funções para ler o arquivo e cria as tabela de tokens:

```
def read_code_from_file(file_name):
    with open(file_name, 'r') as file:
        code = file.read()
    return code
```

```
def create_token_table(tokens, file_name):
```

```

with open(file_name, 'w') as file:
    for token_type, token, position in tokens:
        file.write(f"Token: {token}, Classe: {token_type}, Posição: {position}\n")

```

E por último a função principal para rodar o código:

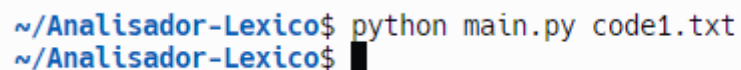
```

if __name__ == "__main__":
    if sys.argv[1] != "code1.txt" and sys.argv[1] != "code2.txt" and sys.argv[1] !=
"code3.txt":
        print("Arquivo nao existe, escolha: code1.txt, code2.txt ou code3.txt")
        exit(1)

    code = read_code_from_file(sys.argv[1])
    analyzer = LexicalAnalyzerFSM()
    tokens = analyzer.tokenize(code)
    create_token_table(tokens, "token_table.txt")

```

Como Solicitado analisador léxico deve recebe o arquivo de textos contendo codigo, na linha de comando conforme Figura 01



```


~/Analizador-Lexico$ python main.py code1.txt
~/Analizador-Lexico$

```

Figura 1 - Print da linha de comando.

Fonte: autores, 2023.

e retornando o token identificado a sua classe e sua posição no arquivo exemplificado na figura 02

 token_table.txt

```
1 Token: if, Classe: KEYWORD, Posição: 0
2 Token: (, Classe: MARKERS, Posição: 3
3 Token: X, Classe: IDENTIFIER, Posição: 4
4 Token: >, Classe: OPERATOR, Posição: 6
5 Token: 12, Classe: NUMBER, Posição: 8
6 Token: ), Classe: MARKERS, Posição: 10
7 Token: {, Classe: MARKERS, Posição: 11
8 Token: WriteSerial, Classe: KEYWORD, Posição: 14
9 Token: (, Classe: MARKERS, Posição: 25
10 Token: s3, Classe: IDENTIFIER, Posição: 26
11 Token: ,, Classe: MARKERS, Posição: 28
12 Token: 2301, Classe: NUMBER, Posição: 30
13 Token: ), Classe: MARKERS, Posição: 34
14 Token: }, Classe: MARKERS, Posição: 37
15
```

Figura 2 - Arquivo de saída Analisador Léxico.

Fonte: autores, 2023.

2 FASE 2 – VERIFICAÇÃO DE CÓDIGO – ANÁLISE SINTÁTICA E SEMÂNTICA

Nesta fase você deverá implementar um analisador sintático usando um parser, usando LL1, ou LR1 e uma estrutura baseada em cálculo de sequentes para a verificação dos tipos dos identificadores, valores e expressões criadas na sua linguagem.

Para realizar a etapa da análise Sintática foi utilizada a ferramenta ANTLR4 para gerar os códigos em python do analisador sintática para isso foi transportado a gramática já apresentada na fazer para o ANTLR e que ficou assim:

grammar CodeJP;

program: decls stmts;

decls: (decl decls)? ;

decl: type ID ';' ;

type: BASIC ;

stmts: (stmt stmts)? ;

bool: join ('||' join)* ;

join: equality ('&&' equality)* ;

equality: rel (('==' | '!=') rel)? ;

rel: expr (('<' | '<=' | '>=' | '>') expr)? ;

expr: term (('+' | '-') term)* ;

term: unary (('*' | '/') unary)* ;

unary: '!' unary | factor ;

factor: NUM | REAL | 'true' | 'false' | STRING | ID;

stmt: 'if' '(' bool ')' block ('else' block)?

| 'while' '(' bool ')' block

| 'do' block 'while' '(' bool ')' ';' ;

| 'break' ';' ;

| 'WritePort' '(' ID ',' ('true' | 'false' | ID) ')' ';' ;

| 'ReadPort' '(' ID ',' STRING ')' ';' ;

| 'ReadSerial' '(' ID ',' expr ')' ';' ;

| 'WriteSerial' '(' ID ',' (STRING | NUM) ')' ';' ;

```

| 'Writeanalog' '(' ID ',' expr ')' ';'
| 'Readanalog' '(' ID ',' STRING ')' ';'
| 'sleep' '(' expr ')' ';'
| ID '=' expr ';'
;

```

```

block: '{' stmts '}' | stmt ;

```

```

BASIC: 'int' | 'float' | 'char' | 'bool' ;
ID: [a-zA-Z_][a-zA-Z0-9_]* ;
NUM: [0-9]+ ;
REAL: [0-9]+ '.' [0-9]* ;
CHAR: [0-9a-zA-Z!@#$%^&*()_+={}];
STRING: '"' (~["\n\r])* '"' ;
WS: [ \t\r\n]+ -> skip ;

```

Após isso foi implementado as chamda do código para código principal para trazendo as seguintes chamadas:

```

from CodeJPLexer import CodeJPLexer
from CodeJPParser import CodeJPParser
from CodeJPListener import CodeJPListener
from antlr4 import CommonTokenStream, FileStream, ParseTreeWalker

```

```

from antlr4.tree.Trees import Trees

```

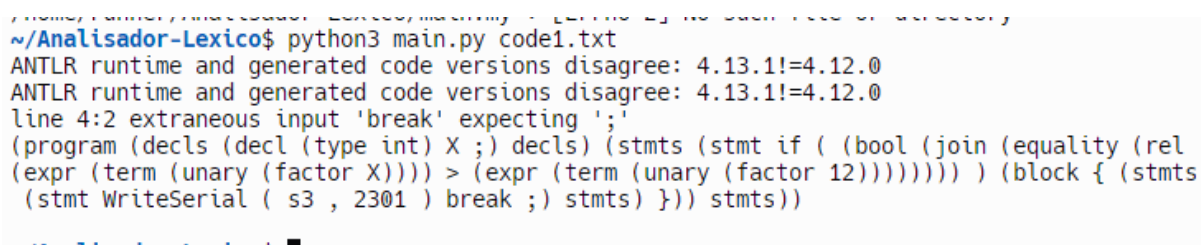
```

input_stream = FileStream(sys.argv[1])
lexer = CodeJPLexer(input_stream)
token_stream = CommonTokenStream(lexer)
parser = CodeJPParser(token_stream)
tree = parser.program()

```

Onde as mesmas fazer as regras de parses e geram a árvore sintática, utilizando o exemplo 1, e tirando o ';' do comando 'break' informa o seguinte erro e apresenta a árvore sintática montada.

```
int X;
if (X > 12) {
    WriteSerial(s3, 2301);
    break
}
```



```
~/Analizador-Lexico$ python3 main.py code1.txt
ANTLR runtime and generated code versions disagree: 4.13.1!=4.12.0
ANTLR runtime and generated code versions disagree: 4.13.1!=4.12.0
line 4:2 extraneous input 'break' expecting ';'
(program (decls (decl (type int) X ;) decls) (stmts (stmt if ( (bool (join (equality (rel
(expr (term (unary (factor X)))) > (expr (term (unary (factor 12))))))) ) (block { (stmts
(stmt WriteSerial ( s3 , 2301 ) break ;) stmts) })) stmts))
```

Figura 3 - Erro sintático.

Fonte: autores, 2023.

Para etapa da análise Semântica foi implementado o seguinte fragmento no código:

```
def are_types_compatible(type1, type2):
    if not variables:
        return False

    if type1.isnumeric():
        return variables[type2] == 'int' or variables[type2] == 'float'

    if type2.isnumeric():
        return variables[type1] == 'int' or variables[type1] == 'float'

    return variables[type1] == variables[type2]

def tokenizeTable(code):
```

```

token_pattern =
r"(\b[a-zA-Z_][a-zA-Z0-9_]*\b|[0-9]+\.[0-9]*|[0-9]+|=|!=|<|=|>|<|>|\+|-|\*|/|=|\\(|\\)|\\{|\\}|\\[\\]
|,|;)"
tokens = re.findall(token_pattern, code)
return tokens

```

```

def Semanticanalyzer(tokens):
    for token in tokens:
        if(token == '+' or token == '-' or token == '*' or token == '/'):
            var1 = tokens[tokens.index(token) - 1]
            op = token
            var2 = tokens[tokens.index(token) + 1]
            if not var1.isnumeric():
                if not is_variable_declared(var1) :
                    print(f"Erro: Variável '{var1}' não foi declarada.")
            if not var2.isnumeric():
                if not is_variable_declared(var2) :
                    print(f"Erro: Variável '{var2}' não foi declarada.")
            if not are_types_compatible(var2, var1) :
                print(f"Erro: Variável '{var2}' e '{var1}' são de tipos diferentes .")

```

```

if(token == '==' or token == '<' or token == '>' or token == '>=' or token == '<='):
    var1 = tokens[tokens.index(token) - 1]
    op = token
    var2 = tokens[tokens.index(token) + 1]
    if not var1.isnumeric():
        if not is_variable_declared(var1) :
            print(f"Erro: Variável '{var1}' não foi declarada.")
    if not var2.isnumeric():
        if not is_variable_declared(var2) :
            print(f"Erro: Variável '{var2}' não foi declarada.")
    if not are_types_compatible(var2, var1) :
        print(f"Erro: Variável '{var2}' e '{var1}' são de tipos diferentes .")

```

```

if(token == '='):
# Divide a linha em partes usando o sinal de igual
    var1 = tokens[tokens.index(token) - 1]
    op = token
    var2 = tokens[tokens.index(token) + 1]

if not is_variable_declared(var1) :
    print(f"Erro: Variável '{var1}' não foi declarada.")
if not are_types_compatible(var1, var2) :
    print(f"Erro: Variável '{var2}' e '{var1}' são de tipos diferentes .")

if(token == 'int' or token == 'float' or token == 'bool' or token == 'char'):
    op = token
    var2 = tokens[tokens.index(token) + 1]
    variables[var2] = op

```

Onde o mesmo separa cada token do código e procura as operações lógicas e matemáticas para verificar os mesmo tipos compatíveis e também verificar se as variáveis apresentadas foram declaradas previamente.

Utilizando o código 1 e tirando as declaração das variáveis apresenta os seguintes erro:

```

~/Analizador-Lexico$ python3 main.py code1.txt
ANTLR runtime and generated code versions disagree: 4.13.1!=4.12.0
ANTLR runtime and generated code versions disagree: 4.13.1!=4.12.0
(program decls (stmts (stmt if ( (bool (join (equality (rel (expr (term (unary (factor X))
)) > (expr (term (unary (factor 12)))))))))) (block { (stmts (stmt WriteSerial ( s3 , 2301
) ;) (stmts (stmt break ;) stmts)) }))) stmts))

Erro: Variável 'X' não foi declarada.
Erro: Variável '12' e 'X' são de tipos diferentes .
~/Analizador-Lexico$ █

```

Figura 4 - Erro semântico.

Fonte: autores, 2023.

3 FASE 3 – GERAÇÃO DE CÓDIGO

Para terceira e última fase foi implementado no compilador um gerador de código Assembly, para ATmega2560, para demonstração com suas funções básicas, para isso foi acrescentado ao código a função 'def create_asm(tokens):' onde dela é gerado o código com base nos tokens coletados no código fonte, segue abaixo a função:

```
def create_asm(tokens):
    cochetes = 0
    contSleep = 0
    contVar = 0
    conf()
    position = 0
    contIf = 0
    for token in tokens:
        if token == 'if':
            contIf += 1
            num1 = tokens[position + 2]
            op = tokens[position + 3]
            num2 = tokens[position + 4]
            lacos[cochetes] = token

            if num1.isnumeric():
                arquivo.write('ldi r16, {}\n'.format(num1))
            if num2.isnumeric():
                arquivo.write('ldi r17, {}\n'.format(num2))

            arquivo.write('rjmp if{}\n'.format(cochetes + contIf))
            arquivo.write('shorif{}\n'.format(cochetes + contIf))
            arquivo.write('jmp end_if{}\n'.format(cochetes + contIf))
            arquivo.write('if{}\n'.format(cochetes + contIf))
            arquivo.write('cp ')
            if num1.isnumeric():
                arquivo.write('r16,')
```

```

else:
    arquivo.write('{}'.format(vareg[num1]))
if num2.isnumeric():
    arquivo.write(' r17\n')
else:
    arquivo.write('{}\n'.format(vareg[num2]))
if op == '==':
    arquivo.write('brne shorif{}\n'.format(cochetes + contlf))
elif op == '>':
    arquivo.write('brlo shorif{}\n'.format(cochetes + contlf))
elif op == '<':
    arquivo.write('brsh shorif{}\n'.format(cochetes + contlf))

cochetes += 1
elif token == 'writeport':

    num1 = tokens[position + 2]
    bool_val = tokens[position + 4]

    isWritePort(num1, bool_val)
elif token == 'readport':
    num1 = tokens[position + 2]
    var = tokens[position + 4]
    isreadPort(num1, var)
elif token == 'while':
    num1 = tokens[position + 2]
    op = tokens[position + 3]
    num2 = tokens[position + 4]
    lacos[cochetes] = token

    arquivo.write('ldi r20, {}\n'.format(num1))
    arquivo.write('ldi r21, {}\n'.format(num2))
    arquivo.write('rjmp while{}\n'.format(cochetes))
    arquivo.write('shorloop{}:\n'.format(cochetes))

```



```

arquivo.write('jmp end_loop{}\n'.format(cochetes))
arquivo.write('while{}\n'.format(cochetes))
arquivo.write('cp r20, r21\n')
if op == '==':
    arquivo.write('brne shorloop{}\n'.format(cochetes))
elif op == '>':
    arquivo.write('brlo shorloop{}\n'.format(cochetes))
elif op == '<':
    arquivo.write('brsh shorloop{}\n'.format(cochetes))
# Continuar com a lógica para o corpo do loop aqui

    cochetes += 1
elif token in ['+', '-', '*']:
    num1 = tokens[position - 1]
    op = token
    num2 = tokens[position + 1]
    IsOperacao(num1, op, num2)

elif token == '}':
    cochetes -= 1
    if lacos[cochetes] == 'if':
        arquivo.write('end_if{}\n'.format(cochetes + contIf))
    elif lacos[cochetes] == 'while':
        arquivo.write('rjmp while{}\n'.format(cochetes))
        arquivo.write('end_loop{}\n'.format(cochetes))

elif token == '=':
    X = tokens[position - 1]
    valor = tokens[position + 1]
    IsAtribuicao(X, valor)
elif token == 'bool' or token == 'int':
    var = tokens[position + 1]
    r = 22 + contVar

```

```

    reg = 'r{}'.format(r)
    vreg[var] = reg
elif token == 'sleep':
    contSleep += 1
    num = tokens[position + 2]
    isSleep(num, contSleep)
    position += 1

```

```

def conf():
    arquivo.write('#define __SFR_OFFSET 0x00\n')
    arquivo.write('#include <avr/io.h>\n')
    arquivo.write('.global program\n')
    arquivo.write('program:\n')
    arquivo.write('ldi r18,0xFF      ;carrega FFh no registrador auxiliar 18\n')
    arquivo.write('out DDRA,r18      ;configura portA como saída\n')
    arquivo.write('out PORTC,r18      ;configura o PC com pull-up interno. inicializa
PC, em low\n')

```

```

def islf(num1, op, num2):
    # Implementação da lógica para a instrução if
    arquivo.write('ldi r10, {}'.format(num1))
    arquivo.write('ldi r12, {}'.format(num2))
    arquivo.write('cp r10, r12\n')
    if op == '==':
        arquivo.write('brne else_block\n')
    elif op == '>':
        arquivo.write('brlo else_block\n')
    elif op == '<':
        arquivo.write('brsh else_block\n')
    # Continuar com a lógica para o bloco if aqui
    arquivo.write('rjmp endif_block\n')
    arquivo.write('else_block:\n')

```



```
def isreadPort(num, var):
    # Implementação da lógica para ler de uma porta

    arquivo.write('readPort{:}\n'.format(num))
    arquivo.write('sbis PINC, {}; Testa o bit correspondente ao pino\n'.format(num))
    arquivo.write('rjmp pinLow      ; Se o bit for 0, o pino está em nível baixo\n')
    arquivo.write('pinHigh:\n')
    arquivo.write('ldi {}, 1 \n'.format(vareg[var]))
    arquivo.write('rjmp EndreadPort{}\n'.format(num))
    arquivo.write('pinLow:\n')
    arquivo.write('ldi {}, 0 \n'.format(vareg[var]))
    arquivo.write('EndreadPort{}\n'.format(num))
```

```
def IsOperacao(num1, op, num2):
    # Implementação da lógica para operações aritméticas
    arquivo.write('ldi r16, {}\n'.format(num1))
    arquivo.write('ldi r17, {}\n'.format(num2))
    if op == '+':
        arquivo.write('add r16, r17\n')
    elif op == '-':
        arquivo.write('sub r16, r17\n')
    elif op == '*':
        arquivo.write('mul r16, r17\n')
```

```
def IsAtribuicao(X, valor):
    # Implementação da lógica para atribuição
    arquivo.write('ldi r16, {}\n'.format(valor))
    arquivo.write('sts {}, r16\n'.format(X))
```

Para demonstração, implementamos novos tres codigos para como exemplo:

```

while (1 > 0){
    writeport( 1 , true);
    sleep(2);
    writeport ( 1, false);
    sleep(2);
}

```

Exemplo 1 é clássico exemplo de pisca pisca.

```
int x;
```

```

while (1 > 0){

readport (1, x);

    if (x == 1) {
        writeport (1, true);
    }
    if (x == 0) {
        writeport (1, false);
    }
}

```

Exemplo 2 é para demonstração de leitura de porta.

```
int x;
```

```

while (1 > 0){
writeport (1, true);
readport (1, x);

    if (x == 1) {
        writeport (1, false);
        sleep(5)
    }
}

```

Exemplo 3 é para demonstração de leitura de porta escrita e timer.

O compilador gera um arquivo com o assembly gerado a partir do exemplos, sobre ATmega2560, utilizamos um arduino 2560, e para gravar utilizamos a própria IDE do arduino.

Link para repositório do GitHub: https://github.com/JeffAbbin/Projeto_Compilador-

Link para rodar o código no replt:

<https://replit.com/@JeffersonAbbin/Analizador-Lexico#main.py>

Link2 para rodar o código no replt: <https://replit.com/join/mjgkqebsex-jeffersonabbin>

REFERÊNCIAS

LINGUAGEM de programação e Compiladores. [S. l.], 1 jan. 2021. Disponível em: <https://frankalcantara.com/Ling-programacao-compiladores/>. Acesso em: 16 abr. 2023.

REGEX refence. [S. l.], 16 abr. 2023. Disponível em: <https://regexr.com/>. Acesso em: 16 abr. 2023.

WHAT is ANTLR?. [S. l.], 25 maio 2023. Disponível em: <https://www.antlr.org>. Acesso em: 25 maio 2023.

ANALISE Léxica. [S. l.], 25 maio 2023. Disponível em: <https://johnidm.gitbooks.io/compiladores-para-humanos/content/part1/lexical-analysis.html>. Acesso em: 25 maio 2023.