
# Relatório

# Processamento de Linguagens

**Alunos:**

**André Freitas – 21112**

**Eduardo Rebelo – 21105**

**Pedro Neves - 21141**

**Professor: Óscar Ribeiro**

**Licenciatura em Engenharia de Sistemas Informáticos**

Barcelos, junho, 2023

# Índice

## Índice de figuras

# 1. Introdução

## 1.1. Contextualização

Este trabalho prático está inserido na unidade curricular de Processamento de Linguagens, do curso de Licenciatura de Engenharia de Sistemas Informáticos.

A realização deste trabalho prático consistiu em implementar uma ferramenta em Python, usando a biblioteca PLY, que interprete uma linguagem capaz de especificar algumas instruções que habitualmente encontramos em qualquer linguagem de programação.

## 1.2. Estrutura do Documento

O relatório deste trabalho prático encontra-se dividido em quatro capítulos:

1. **Introdução** – Neste capítulo, encontra-se um breve resumo sobre o que consiste este trabalho e quais os objetivos da realização deste mesmo;
2. **Implementação** – Na Implementação encontra-se uma descrição completa de todos os pormenores do trabalho, explicando cada fase, assim como o funcionamento do mesmo;
3. **Output e Resultado** – Na análise e testes está representado imagens sobre a execução do programa final, explicando detalhadamente cada passo e ainda uma análise final deste trabalho prático;
4. **Conclusão** - E por fim na conclusão fala sobre o que achamos deste trabalho prático, quer a nível de dificuldades encontradas a meio do projeto e apreciação final sobre o que este trabalho melhorou em nós;

# 2. Implementação

## 2.1 Instrução de escrita

A instrução ESCREVER recebe uma lista de argumentos separados por vírgulas. Os argumentos deste comando podem ser strings (ou seja, sequência de caracteres delimitada por aspas), uma constante numérica inteira, ou outra expressão aritmética.

Segue um exemplo da utilização de instruções de escrita:

escrever "olá mundo!";

escrever "PL ", 2 ,"o ano de", "ESI";

escrever 9+2*3 ;

Para implementar esta instrução precisamos de primeiro declarar a palavra escrever como um token.

```
tokens = ("var", "atribui", "nr", "string", "verdadeiro", "falso",
"nao", "e", "ou", "escrever",
          "entrada", "Inicio", "Fim", "cos", "sen", "se", "fim_se",
"entao", "senao", "para", "de",
          "ate", "fim_para", "fun", "STRING", "aleatorio", "fazer",
"chamada_funcao")
```

*Figura 1: Tokens existentes*

De seguida iremos criar uma regra gramatical que faça a operação desejada:

```
def p_c1(self, p):
    """C : escrever '(' e_list ')' ';'"""
    p[0] = {'op': p[1], 'args': p[3]}
```

*Figura 2: Gramática da instrução escrever*

Por fim temos de fazer todas as verificações necessárias, desde as variáveis, os cálculos numéricos, a identificação do tipo de variável, etc. Isto tudo é agregado nesta grande verificação dentro da zona da sua respetiva instrução.

Esta verificação apesar de ser monótona é bastante complexa, porque é abordado todo o tipo de constante, seja ela medida, frase, número…

```
elif opcode == "escrever":
    message = " ".join(parts[1:]).strip(';')
    variable = parts[1].strip(";")
    if any(char.isdigit() for char in message):
        # Check if the message contains numeric operations
        if re.match(r'^\d+(\s*[-+\/*]\s*\d+)*$', message):
            # Evaluate the numeric operation expression
            try:
                result =
LogicEval.evaluate_expression(message)
                return f'printf("%d", {result});'
            except:
                return "Invalid numeric operation in
escrever instruction"
        else:
            return f'printf("%d", {message});'
    if variable in integer_variables:
        return f'printf("%d", &{variable});'
    elif variable in string_variables:
        return f'printf("%s", {variable});'
    else:
        return f'printf({message});'
```

*Figura 3: Verificações da instrução escrever*

De maneira a poder ser mais fácil de entender como é este processo aqui está a árvore de sintaxe para a instrução de escrever:

```
Escrever
    |
  String
"olá mundo!"
```

*Figura 4: Árvore de sintaxe da função escrever*

## 2.2 Declaração de variáveis

Uma declaração é iniciada pela palavra reservada var apresentando de seguida uma lista de variáveis separadas por uma vírgula, as quais podem ser inicializadas com uma constante inteira, ou uma string.

Segue aqui uma lista de exemplos:

var a = 10

f = 3

A primeira variável vai ser processada corretamente pois contém o identificador correto de uma variável, enquanto a segunda variável irá resultar num erro, sendo que este erro é comunicado para o utilizador como podemos ver em baixo:

```
Error: Variable "dia" is not declared.
```

*Figura 5: Variável não declarada*

No ficheiro lexer.py iremos declarar como funciona a sua sintaxe, usando uma expressão regular de maneira a poder a adicionar na nossa gramática:

```
def t_var(self, t):
    r"[a-z_]+"
    return t
```

*Figura 6: Deteção da instrução*

Depois fazemos a gramática envolvida nesta instrução:

```
def p_e1(self, p):
    """ E : var """
    p[0] = {'var': p[1]}
```

*Figura 7: Gramática da variável*

```
def p_var_list(self, p):
    """ var_list : var
                 | var_list ',' var """
    if len(p) == 2:
        p[0] = [p[1]]
    else:
        p[0] = p[1]
        p[0].append(p[3])
```

*Figura 8: Lista de variáveis*

## 2.3 Atribuição a uma variável

Uma atribuição passa por atribuir a uma variável: uma expressão aritmética, o resultado de leitura de um valor introduzido pelo utilizador, ou a geração aleatória de um valor. Uma expressão aritmética poderá ser atribuídas a uma variável, havendo ainda a possibilidade de utilizar os operadores para incrementar a variável. Interessa verificar se as variáveis utilizadas foram previamente declaradas.

Segue aqui uma lista de exemplos:

tmp = 7+3 ;

a = 10 (30 + tmp ) ;

a ++; a += 10; b = tmp * (a + 10);

Começamos por atribuir a sintaxe por trás, identificado o que vai ser o nome da nossa variável e o seu respetivo, valor:

```
def _atribui(args):  # A=10   {'op':'atr'  'args': [ "A", 10 ]}  =>  _attrib( [ 'A', 10 ] )
    varid = args[0]  # 'A'
    value = args[1]  # 10
    LogicEval.symbols[varid] = value  # symbols { 'A':10 }
    return None
```

*Figura 9: Sintaxe da atribuição de uma variável*

Dada a sintaxe é feita a respetiva gramática no ficheiro gramar.py:

```
def p_a1(self, p):
    """ A : var atribui E ';' """
    p[0] = {"op": "atribui", "args": [p[1], p[3]]}
```

*Figura 10: Gramática da atribuição de uma variável*

Como é pedido no enunciado também temos de ter oportunidade de ter um valor chamado “aleatorio()” que irá atribuir um número aleatório entre 0 e o número introduzido.

```
def t_aleatorio(self, t):
    r"""aleatorio"""
    return t
```

*Figura 11: Sintaxe da instrução aleatorio*

```
def p_a2(self, p):
    """A : var aleatorio '(' E ')' ';'"""
    p[0] = {"op": "aleatorio", "args": [{"var": p[2]},
p[4]]}
```

*Figura 12: Gramática da instrução aleatorio*

A nossa verificação vai englobar todas estas restrições e vai verificar o nosso input de maneira correta. Como esta função é demasiado grande vamos apenas mostrar as partes relevantes.

```
if variable_value.startswith("aleatorio(") and
variable_value.endswith(");"):
    # Extract the range from the variable value
    range_value = variable_value[10:-2]  # Extract the
range value from "aleatorio(range)"

    # Check if the range value is a valid integer
    if not range_value.isdigit():
        return "Invalid range value: must be an integer"

    # Generate a random number within the given range
    variable_value = random.randint(0, int(range_value))
elif variable_value.startswith('"') and
variable_value.endswith('"'):
    # Text variable
    variable_value = variable_value[1:-1]
    string_variables.append(variable_name)
else:
    # Check if the variable value is a numeric operation
    if re.match(r'^\d+(\s*[-+\/*]\s*\d+)*$',
variable_value):
        # Evaluate the numeric operation expression
        try:
            if assignment_operator == "+=":
                # Retrieve the previously stored value
                previous_value = storage.get(variable_name,
0)
                variable_value =
LogicEval.evaluate_expression(f"{previous_value} +
({variable_value})")
            else:
                variable_value =
LogicEval.evaluate_expression(variable_value)
                integer_variables.append(variable_name)
        except:
            return "Invalid variable value: unable to
evaluate numeric operation"
```

*Figura 13: Verificações da atribuição de variáveis*

Este processo todo pode ser demonstrado de maneira simples com a visualização da árvore de sintaxe.

```
VariableDeclaration
         |
      Variable
      /   |   \
  Name   Oper Value
  "x"    "="     10
```

*Figura 14: Árvore sintática da instrução de variável*

É importante ainda falar que também foi criado um método de leitura chamado “entrada()” como foi pedido nesta instrução de variável.

Aqui está a sua gramática:

```
def p_c2(self, p):
    """C : entrada '(' var ')' ';'"""
    p[0] = {'op': 'attrib', 'args': [{'var': p[1]}, p[3]]}
```

*Figura 15: Gramática da leitura de variáveis*

Além disto temos as verificações necessárias para o tipo de variável a ser indicado.

```
elif opcode == "entrada":
    variable = parts[1].strip(";")

    # Check if the variable is an integer or string
    if variable in integer_variables:
        return f'scanf("%d", &{variable});'
    elif variable in string_variables:
        return f'scanf("%s", {variable});'
    else:
        return ""
```

*Figura 16: Verificações de instrução de leitura*

## 2.4 Ciclos

A linguagem de expressões aritméticas também suporta ciclos, como qualquer linguagem imperativa, com seguinte sintaxe:

PARA i EM [10..20] FAZER < instruções > FIM PARA ;

Este comando irá executar 11 vezes as instruções incluídas no ciclo, sendo que em cada iteração o valor da variável **i** irá sendo alterado, começando em 10, e incrementando até 20.

Para a implementação desta instrução começamos por definir a gramática:

```
def p_c3(self, p):
    """ C : para var de E ate E fazer c_list  fim_para """
    p[0] = {
        "op": "para",
        "args": [p[2], p[4], p[6]],
        "data": p[8],
        "fazer": True
    }
```

*Figura 17: Gramática da instrução "para"*

Após isto teremos a sua avaliação:

```
def _para(args):
    var, start_value, end_value = args
    start_value =
LogicEval.evaluate_expression(start_value)
    end_value = LogicEval.evaluate_expression(end_value)
    for i in range(start_value, end_value + 1):
        LogicEval.symbols[var] = i
    return None
```

*Figura 18: Avaliação do ciclo*

Apesar de ser referenciado numa pequena nota no enunciado não foi possível implementar ciclos dentro de ciclos devido a restrições de tempo.

Antes de mostrar a verificação de todo o input, vou demonstrar a árvore de sintaxe desta instrução de maneira a facilitar a sua compreensão:

```
          para
       /    |    \
variable start  end
                  |
              instruction
```

*Figura 19: Árvore de sintaxe da instrução "para"*

Aqui encontram-se todas as verifica ao input desta instrução:

```
elif opcode == "para":
    loop_variable, start_value, end_value =
parts[1].strip("("), parts[3].strip("("), parts[5].strip("(
)")
    instruction = " ".join(parts[8:]).strip(";")
    loop_output = f'for (int {loop_variable} =
{start_value}; {loop_variable} <= {end_value};
{loop_variable}++)' + ' {\n'
    fazer_body = parts[parts.index("fazer") + 1:] if
"fazer" in parts else []
    message = " ".join(parts[parts.index("fazer") + 2:])
    try:
        start_value = int(start_value)
        end_value = int(end_value)
        for i in range(start_value, end_value + 1):
            LogicEval.symbols[loop_variable] = i
            for instruction in fazer_body:
                if instruction.startswith("escrever"):
                    if re.match(r'^\d+(\s*[-
+\/*]\s*\d+)*$', message):
                        loop_output += "\t\tprintf(%d," +
LogicEval.evaluate_expression(message) + '\n'
                    else:
                        loop_output += "\t\tprintf(%s," +
message + ');\n'
                if instruction.startswith("entrada"):
                    variable = parts[parts.index("fazer") +
2:]
                    for var in variable:
                        # Check if the variable is an
integer or string
                        if var in integer_variables:
                            loop_output +=
f'\t\tscanf("%d", &{var});\n'
                        if var in string_variables:
                            loop_output +=
f'\t\tscanf("%s", {var});\n'
        loop_output += '\t}'
        return loop_output
    except ValueError:
        loop_output = f"Invalid start or end value in para
loop: {start_value}, {end_value}"
    return loop_output
```

*Figura 20: Avaliação do input da instrução "para"*

## 2.5 Funções

A nossa implementação desta instrução passou por ser relativamente parecida à execução de uma função em grande parte das linguagens de programação. Aqui se segue um exemplo de entrada para esta instrução:

**funcao soma (a, b)**

**var c = a+b**

**escrever c;**

**fim;**

**soma(3,4)**

Houve um grande duvida de como se ia implementar esta função e a sua respetiva chamada, mas no final acabamos for definir esta instrução da seguinte maneira.

Começando pela sintaxe iremos definir a estrutura de ambos a própria função e a sua respetiva chamada:

```
def _funcao(args):
    function_name = args[0]
    var_list = args[1]
    code = args[2]
    LogicEval.functions[function_name] = {"var_list":
var_list, "code": code}
```

*Figura 21: Criação da sintaxe da função*

```
def t_chamada_funcao(self, t):
    r"[a-z_]+\s*\("
    t.value = t.value.strip()[:-1]  # Remove the opening parenthesis
    t.type = "chamada_funcao"
    return t
```

*Figura 22: Criação da sintaxe da chamada de função*

Após isto foi feita a respetiva gramática:

```
def p_a0(self, p):
    """ A : fun var '(' args ')' '{' code '}' """
    p[0] = {'op': 'fun', 'args': [{'var': p[2]}, p[4]],
"code": p[7]}
```

*Figura 23: Criação da gramática da função*

```
def p_e4(self, p):
    """ E : chamada_funcao arg_list
          | chamada_funcao """
    arg_list = [] if len(p) == 2 else p[2]
    p[0] = {"op": "call", "args": [{"var": p[1]}] + arg_list}
```

*Figura 24: Criação da gramática da chamada de função*

Por último temos as verificações de lógica feitas no ficheiro “eval.py”:

```
def _funcao(args):
    function_name = args[0]
    var_list = args[1]
    code = args[2]
    LogicEval.functions[function_name] = {"var_list": var_list,
"code": code}
```

*Figura 25: Verificação de lógica para a funções*

```
def _call(args):
    name, arg_list = args
    if name not in LogicEval.functions:
        raise Exception(f"Function not defined: {name}")
    if len(arg_list) !=
len(LogicEval.functions[name]["var_list"]):
        raise Exception(f"Function called with the wrong number
of arguments: {name}")
    for var, value in zip(LogicEval.functions[name]["var_list"],
arg_list):
        LogicEval.symbols[var] = value
    result = LogicEval.eval(LogicEval.functions[name]["code"])
    for var in LogicEval.functions[name]["var_list"]:
        del LogicEval.symbols[var]
    return result
```

*Figura 26: Avaliação lógica da chamada de funções*

Mais uma vez por causa do pouco tempo não foi possível fazer as verificações de input para esta componente do trabalho, pedimos desculpas.

## 2.6 Comentários

A implementação dos comentários foi bastante simples. Apenas meia dúzia de verificações chegou para completar esta instrução toda. E quando falamos em comentários falamos em comentários de uma linha só **(//)** e de múltiplas linhas **(/*…*/)**.

Começando pela sintaxe:

```
def t_comment(self, t):
    r"""\#.*"""
    t.value = t.value[1:]
    if '\n' not in t.value:
        t.type = 'comment_single'
        t.value = f"//{t.value}"
    else:
        t.type = 'comment_multi'
        t.value = f"/*{t.value}*/"
    return t
```

*Figura 27: Sintaxe da instrução de comentários*

Prosseguindo para a gramática onde abordamos as duas maneiras de comentar e o conteúdo desses comentários:

```
def p_comment_single(self, p):
    """comment_single : '#' comment_body"""
    p[0] = {'comment': "//" + p[2]}

def p_comment_multi(self, p):
    """comment_multi : '#' '*' comment_body '*' '#'"""
    p[0] = {'comment': "/*" + p[3] + "*/"}

def p_comment_body(self, p):
    """comment_body : STRING
                    | comment_body STRING"""
    if len(p) == 2:
        p[0] = p[1]
    else:
        p[0] = p[1] + p[2]
```

*Figura 28: Gramática da implementação de comentários*

Por fim vamos ter duas verificações quanto ao input:

```
# Skip single-line comments
if instruction.startswith("//"):return ""
```

```
# Skip multi-line comments
if "/*" in instruction and "*/" in instruction:return ""
```

*Figura 29: Verificações da entrada de comentários*

## 2.7 Uso do condicional

A criação desta implementação também deu muito que falar dado que no enunciado não tínhamos nenhuma informação à cerca desta instrução. Decidimos abordar nos caminhos da instrução “para” apesar de não se tratar de um ciclo.

A sintaxe não foi necessário definir além de colocar as palavras necessárias na lista de tokens da Figura 1: Tokens existentes.

Mas no que toca à gramática já foi necessário intervirmos, criando os componentes necessários para esta instrução:

```
def p_s(self, p):
    """S : C
         | E
         | A
         | condicao"""
    p[0] = p[1]

def p_condicao(self, p):
    """ condicao : se E entao c_list senao c_list fim_se
"""
    p[0] = {
        "op": "se", "senao"
        "args": [p[2]],
        "data": [p[4], p[6]],
        "entao": True
    }
```

*Figura 30: Gramática necessário para a instrução "se"*

Após isto tivemos de criar uma grande verificação para possibilitar o processamento da entrada desta instrução, abordado tudo sobre esta instrução, desde a condição à instrução.

```
@staticmethod
def _se(cond, entao, senao):
    return LogicEval.eval(entao if cond else senao)
```

*Figura 31: Verificação lógica do uso do condicional*

Antes de mostrarmos esta avaliação, gostávamos de apresentar a arvore de sintaxe desta instrução:

```
            se
         /   |   \
condition entao senao
            |
        instruction
```

*Figura 32: Árvore sintática*

Esta próxima verificação, portanto, iremos só mostrar o relevante em consideração ao leitor:

```
elif opcode == "se":
    condition = " ".join(parts[1:parts.index("entao")])
    entao_body = parts[parts.index("entao") +
1:parts.index("senao")] if "senao" in parts else
parts[parts.index("entao") + 1:]
    senao_body = parts[parts.index("senao") + 1:] if "senao" in
parts else []
    try:
        if LogicEval.evaluate_condition(condition):
            output = 'if ' + condition + ' {\n\t\t'
            output_added = False  # Flag to track if any output
was added
            message = " ".join(parts[parts.index("entao") +
2:parts.index("senao")])
            for instruction in entao_body:
                if instruction.startswith("escrever"):
                    if re.match(r'^\d+(\s*[-+\/*]\s*\d+)*$',
message):
                        output += "printf(%d," +
LogicEval.evaluate_expression(message) + '\n\t\t'
                        output_added = True  # Set the flag to
True
                    else:
                        output += "printf(%s," + message + ');\n'
                        output_added = True  # Set the flag to
True
         if not output_added:
        output += '// Add default output or alternative
instructions\n'
    output += '\t}'
    return output
else:
    output = 'else {\n\t\t'
    output_added = False  # Flag to track if any output was added
    message = " ".join(parts[parts.index("senao") + 2:])

     f not output_added:
            output += '\t// Add default output or alternative
instructions\n'
        output += '\t}'
        return output
except:
    return "Invalid condition in se instruction"
```

*Figura 33: Avaliação da entrada da instrução "se"*

# 3. Output e Resultados

Para este capítulo vamos fazer uma coisa muito simples que é percorrer cada instrução, apresentar o input introduzido e output criado a partir dessa instrução.

Além disso também vamos abordar a capacidade do projeto ter duas compilações diferentes dependendo do objetivo do utilizador.

## 3.1 Instrução de escrita

### Input:

```
escrever "olá mundo!";
escrever 9+2*3;
```

*Figura 34: Input da instrução "escrever"*

### Output:

```
printf("olá mundo!");
printf("%d", 15);
```

*Figura 35: Output da instrução “escrever"*

## 3.2 Declaração de variáveis

### Input:

```
var numero = 2023
var mes = "maio"
```

*Figura 36: Input da instrução "var"*

### Output:

```
Variable 'numero' set to 2023
Variable 'mes' set to maio
```

*Figura 37: Output da instrução "var"*

## 3.3 Atribuição a uma variável

### Input:

```
var tmp = 7+3
var a = 10*30+8
var a += 8
var aleatorio = aleatorio(10);
var aleatorio2 = aleatorio(100);
f = 3;
```

*Figura 38: Input para a atribuição de variáveis*

### Output:

```
Variable 'tmp' set to 10
Variable 'a' set to 308
Variable 'a' set to 316
Variable 'aleatorio' set to 8
Variable 'aleatorio2' set to 33
Error: Variable "f" is not declared.
```

*Figura 39: Output da atribuição de variáveis*

## 3.4 Ciclos

### Input:

```
para (i de 14 ate 20) fazer escrever "Estou a funcionar"
```

*Figura 40: Input da instrução "para"*

### Output:

```
for (int i = 14; i <= 20; i++) {
                printf(%s,"Estou a funcionar");
                printf(%s,"Estou a funcionar");
                printf(%s,"Estou a funcionar");
                printf(%s,"Estou a funcionar");
                printf(%s,"Estou a funcionar");
                printf(%s,"Estou a funcionar");
                printf(%s,"Estou a funcionar");
        }
```

*Figura 41: Output da instrução "para"*

## 3.5 Funções

### Input:

```
funcao maisdez (c)
    var c += 10
    escrever c;
    fim;

call maisdez (c);
```

*Figura 42: Input da instrução "funcao"*

### Output:

```
void maisdez(c){
    Variable 'c' set to 10
    printf(c);
}

maisdez(c);
```

*Figura 43: Output da instrução "funcao"*

## 3.6 Comentários

### Input:

```
//isto é um comentário
/*isto
também é
um comenário*/
```

*Figura 44: Input da instrução "comentario"*

### Output:

Como o objetivo é ignorar os comentários estes não vão ser apresentados.

```

```

*Figura 45: Output da instrução "comentario"*

## 3.7 Uso do condicional

### Inputs:

```
var numero = 2023
se (numero > 80) entao escrever "Positivo" senao entrada a
```

```
var numero = 5
se (numero > 80) entao escrever "Positivo" senao entrada a
```

*Figura 46: Inputs da instrução "se"*

### Output:

```
Variable 'numero' set to 2023
if (numero > 80) {
        printf(%s,"Positivo");
        }
```

```
Variable 'numero' set to 5
else {
        scanf("%d", &numero);
        }
```

*Figura 47: Outputs da instrução "se"*

## 3.8 Duas maneiras de compilar

Como era dito o enunciado além de processar estas instruções todas era necessário apresentar estras instruções em formato de linguagem C. Para lidar com esta possibilidade damos a opção ao utilizador de fazer o que quer.
Para compilar e resolver todas as instruções o utilizador deverá colocar no terminal o script:

```
python main.py --file entrada.ea.txt --mode console
```

*Figura 48: Script de processamento das instruções*

Para ler todas as instruções e traduzi-las para um ficheiro c deverá usar este script:

```
python main.py --file entrada.ea.txt --mode c
```

*Figura 49: Script de tradução para a linguagem C*

Este script vai criar um ficheiro chamado **"programa.c"** e ele vai ser alguma coisa como isto:

```c
#include <stdio.h>

int main() {
    /* Generated instructions */
    printf("olá mundo!");
    printf("%d", 9+2*3);
```

*Figura 50: Código do ficheiro de output*

# 4. Conclusão

Deparamo-nos com um trabalho bastante desafiador e complicado, uma vez mais a pesquisa por respostas e a entre ajuda foram chaves fundamentais para a implementação deste código, achamos que sem dúvida foi um trabalho gratificante e que poderá impulsionar-nos em várias vertentes na medida em que a Linguagem Python é, atualmente, das mais usadas e que se procura no mercado de trabalho.

Para a produção deste trabalho, tivemos de explorar e aprofundar-nos em matérias relacionadas com a linguagem Python e C, coisa que numa visão do grupo é bastante benéfico e enriquecedor pois ao mesmo tempo que exigiu conhecimento das aulas e o código do professor foi também necessário trazer ao de cima o nosso espírito investigador uma vez que foi crucial a busca de informação a diversos tipos de plataformas.

Infelizmente não foi possível implementar tudo aquilo que queríamos por questões de tempo, mas mesmo achamos que fizemos um projeto de qualidade e que devíamos estar orgulhosos.