Relatório do trabalho da disciplina de Processamento de Linguagens

Trabalho Prático 2

Rodrigo Pilar - 26536

Rúben Ribeiro - 25994

Licenciatura em Engenharia de Sistemas Informáticos

junho de 2024

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Afirmo por minha honra que não recebi qualquer apoio não autorizado na realização deste trabalho prático. Afirmo igualmente que não copiei qualquer material de livro, artigo, documento web ou de qualquer outra fonte exceto onde a origem estiver expressamente citada.

Rodrigo Pilar - 26536

Rúben Ribeiro - 25994

Índice

Introdução

Este relatório apresenta o trabalho prático desenvolvido no âmbito da unidade curricular Processamento de Linguagens com o objetivo de implementar uma aplicação em Python utilizando a biblioteca PLY. A aplicação em questão interpreta uma linguagem funcional, capaz de especificar instruções comuns encontradas em linguagens de programação funcionais.

O principal propósito deste trabalho foi desenvolver uma aplicação que seja capaz de processar um arquivo de texto com extensão .fca, contendo uma sequência de comandos para especificar a linguagem funcional. A aplicação deve ser capaz de aplicar esses comandos de forma a calcular o resultado desejado. Além disso, foi considerada a possibilidade de gerar um arquivo com a implementação correspondente em código C.

O relatório abordará detalhadamente o processo de desenvolvimento da aplicação, desde a análise dos requisitos até a implementação das funcionalidades necessárias. Serão discutidas as decisões de implementação e os desafios enfrentados ao longo do desenvolvimento do projeto.

Por fim, serão apresentados os resultados obtidos com a aplicação, incluindo exemplos de utilização e possíveis melhorias ou extensões que poderiam ser implementadas no futuro. Este trabalho representa não só a aplicação prática dos conceitos aprendidos na disciplina, mas também uma oportunidade para aprofundar o conhecimento em linguagens de programação funcionais e em técnicas de interpretação de linguagens.

Descrição lexer.py

Definição dos tokens simples

```
#Lista de nomes dos tokens
tokens = (
'NUMBER', # Número
'PLUS', # +
'MINUS', # -
'TIMES', # *
'DIVIDE', # /
'LPAREN', # (
'RPAREN', # )
'SEMICOLON', # ;
'ASSIGN', # =
'IDENTIFIER', # Identificador
'STRING', # Cadeia de caracteres
'CONCAT', # <>
'WRITE', # Palavra reservada ESCREVER
'READ', # Palavra reservada ENTRADA
'RANDOM', # Palavra reservada ALEATORIO
'FUNCTION', # Palavra reservada FUNCAO
'END', # Palavra reservada FIM
'COMMA', # ,
'LBRACKET', # [
'RBRACKET', # ]
'COMMENT', # Comentário
'COLON', # :
'MAIOR', # /\
'MENOR', # \/
)
```

Neste trecho, são definidos os nomes dos tokens que o analisador léxico será capaz de reconhecer.
Este trecho define uma lista de nomes de tokens que o analisador léxico será capaz de reconhecer. Cada nome representa um tipo de elemento sintático na linguagem, como operadores, delimitadores, palavras-chave, identificadores, strings e números.

Regras de expressão regular para os tokens simples

```
# Regras de expressão regular para tokens simples
t_PLUS = r'\+' # +
t_MINUS = r'-' # -
t_TIMES = r'\*' # *
t_DIVIDE = r'/' # /
t_LPAREN = r'\(' # (
t_RPAREN = r'\)' # )
t_SEMICOLON = r';' # ;
t_ASSIGN = r'=' # =
t_CONCAT = r'<>' # <>
t_COMMA = r',' # ,
t_LBRACKET = r'\[' # [
t_RBRACKET = r'\]' # ]
t_COLON = r':' # :
```

Estas expressões regulares correspondem a tokens simples, que são operadores aritméticos, símbolos de pontuação e outros caracteres usados na linguagem. Cada expressão regular associa um padrão específico de caracteres a um nome de token.

Definição dos Tokens de comparação

```
def t_MAIOR(t):
    r'/\\'                  # /\
    return t

def t_MENOR(t):
    r'\\/'                  # \/
    return t
```

Estas funções definem as expressões regulares para os operadores de comparação **MAIOR** e **MENOR**, permitindo que o analisador léxico reconheça e categorize esses operadores corretamente durante a análise do código.

Funções para identificadores e palavras-chave reservadas

```
# Nomes de funções reservadas
reserved = {
'ESCREVER': 'WRITE', # Palavra reservada ESCREVER
'ENTRADA': 'READ', # Palavra reservada ENTRADA
'ALEATORIO': 'RANDOM', # Palavra reservada ALEATORIO
'FUNCAO': 'FUNCTION', # Palavra reservada FUNCAO
'FIM': 'END', # Palavra reservada FIM
}

# Identificadores e palavras reservadas
def t_IDENTIFIER(t):
r'[a-zA-Z_][a-zA-Z0-9_]*[\?\!]?' # Identificadores
t.type = reserved.get(t.value, 'IDENTIFIER') # Verifica se é palavra reservada
return t
```

Inicialmente definido um dicionário chamado **reserved**, que mapeia palavras reservadas da linguagem para tokens específicos. Estas palavras reservadas incluem comandos como **ESCREVER**, **ENTRADA**, **ALEATORIO**, **FUNCAO** e **FIM**, que são mapeados para os tokens **WRITE**, **READ**, **RANDOM**, **FUNCTION** e **END**, respetivamente.

Este mapeamento é crucial para que o analisador léxico reconheça corretamente estes comandos durante a análise do código.

A função **t_IDENTIFIER** usa uma expressão regular para reconhecer identificadores, que devem começar com uma letra ou sublinhado e podem conter letras, dígitos, sublinhados e opcionalmente terminar com '?' ou '!'. A função também verifica se o identificador corresponde a uma palavra-chave reservada (definida no dicionário reserved) e, se for o caso, atribui o tipo de token apropriado.

Funções para strings e strings interpoladas

```
# Cadeias de caracteres
def t_STRING(t):
r'\"([^\\\n]|(\\.))*?\"' # Expressão regular para cadeias de caracteres
t.value = t.value[1:-1] # Remove as aspas duplas
return t
```

A função **t_STRING** reconhece cadeias de caracteres delimitadas por aspas duplas que podem conter caracteres de escape. Por exemplo, "olá", "mundo", e "string com espaços". A função remove as aspas duplas antes de retornar o token.

Função para comentários

```
# Comentários
def t_COMMENT(t):
r'\-\-.*|\{-[^-]*-\}' # Expressão regular para comentários
pass # Ignora o comentário
```

A função **t_COMMENT** reconhece comentários na linguagem, que podem ser de linha única (iniciados por --) ou de múltiplas linhas (delimitados por {}), e ignora-os, ou seja, não gera tokens para comentários.

Função para números

```
# Números
def t_NUMBER(t):
r'\d+' # Expressão regular para números
t.value = int(t.value) # Converte o valor para inteiro
return t
```

A função **t_NUMBER** reconhece números inteiros e converte a sequência de caracteres correspondente num valor inteiro antes de retornar o token.

Ignorar caracteres em branco

```
# Caracteres ignorados (espaços e tabulações)
t_ignore = ' \t'
```

A variável **t_ignore** informa ao analisador léxico para ignorar espaços em branco e tabulações, que não são relevantes para a análise sintática.

Tratamento de quebras de linha

```
# Quebras de linha
def t_newline(t):
r'\n+' # Expressão regular para novas linhas
t.lexer.lineno += t.value.count("\n") # Incrementa o número de linhas
```

A função **t_newline** trata dos parágrafos, contando o número de novas linhas e atualizando o número da linha atual no analisador léxico. Isso é útil para rastrear a localização dos tokens no código fonte.

Regra de tratamento de erros

```
# Regra de tratamento de erros
def t_error(t):
print(f"Carácter ilegal '{t.value[0]}'") # Imprime o carácter ilegal
t.lexer.skip(1) # Ignora o carácter ilegal
```

A função **t_error** é chamada quando um caractere inválido é encontrado. Ela imprime uma mensagem de erro e ignora o caractere inválido, avançando o analisador léxico para o próximo caractere.

Construção do analisador léxico

```
# Constrói o lexer
lexer = lex.lex()
```

Esta linha de código constrói o analisador léxico a partir das regras e funções definidas anteriormente, preparando-o para ser utilizado na análise de código fonte escrito na linguagem específica.

Descrição grammar.py

Importações e Preparações Iniciais

```
import ply.yacc as yacc
import random
import re
from lexer import tokens # Import tokens from lexer
from lexer import lexer # Import lexer
```

Importamos a biblioteca **ply.yac** para construir o **parser**, além de outras bibliotecas necessárias como **random** e **re** para funções aleatórias e de substituição de padrões. Os **tokens** e o **lexer** são importados de um módulo lexer previamente definido (lexer.py).

```
# Regras de precedência
precedence = (
('left', 'CONCAT'),
('left', 'PLUS', 'MINUS'),
('left', 'TIMES', 'DIVIDE'),
('left', 'MAIOR', 'MENOR'),
)
```

Define a precedência dos operadores. Operadores de concatenação (`<>´) têm menor precedência que os operadores aritméticos (`+´,` -´, `*´, `/´).

Estruturas de Dados para Armazenamento de Variáveis e Funções

```
# Dicionário de nomes (para armazenar variáveis)
names = {}

# Dicionário de funções (para armazenar definições de funções)
functions = {}
```

Dois dicionários são usados para armazenar variáveis (`names´) e definições de funções (`functions´).

Função de Interpolação de Strings

```
# Função para manipular a interpolação de strings
def interpolate_string(s):
def replace_var(match):
var_name = match.group(1)
return str(names.get(var_name, f'#{var_name}'))
return re.sub(r'#\{(\w+)\}', replace_var, s)
```

A função **interpolate_string** substitui variáveis dentro de uma string interpolada (delimitadas por `#{}´) pelos seus valores armazenados no dicionário `names´.

Regras de Parsing

Estrutura do Programa

```
# Estrutura do programa
def p_program(p):
'program : statement_list'
p[0] = p[1]
```

A função **p_program** define a regra de produção para o programa, que é uma lista de instruções (`statement_list).

Lista de Declarações

```
def p_statement_list(p):
'''statement_list : statement_list statement
| statement'''
if len(p) == 3:
p[0] = p[1] + [p[2]]
else:
p[0] = [p[1]]
```

A função **p_statement_list** define a regra de produção para uma lista de instruções, que pode ser uma instrução única ou múltiplas instruções.

Declarações

```
def p_statement(p):
'''statement : assignment SEMICOLON
| write_statement SEMICOLON
| read_statement SEMICOLON
| random_statement SEMICOLON
| function_definition
| function_call SEMICOLON'''
p[0] = p[1]
```

A função **p_statement** utiliza a sintaxe da biblioteca **ply** (Python Lex-Yacc) para definir as diferentes formas que uma instrução pode assumir. Esta definição de regra de produção assegura que o analisador sintático reconheça e processe corretamente as diferentes formas de instruções na linguagem, garantindo que a sintaxe do código seja válida e permitindo a interpretação e execução adequada do mesmo.

Atribuição

```
def p_assignment(p):
'assignment : IDENTIFIER ASSIGN expression'
names[p[1]] = eval_expression(p[3])
p[0] = ('assign', p[1], p[3])
```

A função **p_assignment** executa uma atribuição, onde um identificador recebe o valor de uma expressão. O valor é armazenado no dicionário `names´ Esta função ensina ao analisador como reconhecer uma instrução do tipo “x = 5”, calcular o valor da expressão à direita do “=”, armazenar este valor na variável “x”, e criar uma representação interna dessa operação para uso futuro..

Declaração de Escrita

```
def p_write_statement(p):
'write_statement : WRITE LPAREN expression RPAREN'
p[0] = ('write', p[3])
```

A função **p_write_statement** define uma declaração de escrita, onde o valor de uma expressão é impresso.

Declaração de Leitura

```
def p_read_statement(p):
'read_statement : IDENTIFIER ASSIGN READ LPAREN RPAREN'
value = input("Introduza um valor: ")
names[p[1]] = value
p[0] = ('read', p[1], value)
```

A função **p_read_statement** estabelece uma declaração de leitura, onde um valor é lido do input do utilizador e atribuído a uma variável.

Declaração de Número Aleatório

```
def p_random_statement(p):
'random_statement : IDENTIFIER ASSIGN RANDOM LPAREN NUMBER RPAREN'
value = random.randint(0, p[5])
names[p[1]] = value
p[0] = ('random', p[1], value)
```

A função **p_random_statment** indica uma declaração de número aleatório, onde um número aleatório entre 0 e um valor especificado é gerado e atribuído a uma variável.

Definição de Função

```
def p_function_definition(p):
'''function_definition : FUNCTION IDENTIFIER LPAREN parameters RPAREN COMMA COLON expression
SEMICOLON
| FUNCTION IDENTIFIER LPAREN parameters RPAREN COLON statement_list END'''
if len(p) == 10:
functions[p[2]] = (p[4], [p[8]])
p[0] = ('function_def', p[2], p[4], p[8])
else:
functions[p[2]] = (p[4], p[7])
p[0] = ('function_def', p[2], p[4], p[7])

def p_function_call(p):
'function_call : IDENTIFIER LPAREN arguments RPAREN'
p[0] = ('function_call', p[1], p[3])
```

A função **p_function_definition** executa a criação de funções, que podem ter um único corpo de expressão ou uma lista de declarações.

Chamadas de Função

```
def p_function_call(p):
'function_call : IDENTIFIER LPAREN arguments RPAREN'
p[0] = ('function_call', p[1], p[3])
```

Esta função analisa a chamada de funções na linguagem, onde o identificador da função é seguido por parênteses contendo os argumentos passados. Os argumentos são avaliados e passados para a função correspondente. O resultado da chamada é armazenado como uma tupla contendo o tipo de chamada de função, o identificador da função e os argumentos.

Avaliação de Declarações e Expressões

Avaliação de Declarações

```
def eval_statement(stmt, local_context=None):
if local_context is None:
local_context = names

if stmt[0] == 'assign':
rhs = eval_expression(stmt[2], local_context)
local_context[stmt[1]] = rhs
elif stmt[0] == 'write':
result = eval_expression(stmt[1], local_context)
print(result)
elif stmt[0] == 'read':
local_context[stmt[1]] = stmt[2]
elif stmt[0] == 'random':
local_context[stmt[1]] = stmt[2]
elif stmt[0] == 'function_def':
functions[stmt[1]] = (stmt[2], stmt[3])
elif stmt[0] == 'function_call':
func_name = stmt[1]
args = [eval_expression(arg, local_context) for arg in stmt[2]]
param_names, body = functions[func_name]
saved_context = local_context.copy()
local_context = {}
for param, arg in zip(param_names, args):
local_context[param] = arg
result = None
if isinstance(body, list):
for sub_stmt in body:
result = eval_statement(sub_stmt, local_context)
else:
result = eval_statement(body, local_context)
local_context.clear()
local_context.update(saved_context)
return result
```

Aa função **eval_statement** avalia e executa uma declaração na linguagem. Ela recebe uma declaração **stmt** num contexto local opcional **local_context**, que contém as variáveis e funções definidas anteriormente. Se nenhum contexto local for fornecido, ele usa o contexto global **names**.

Avaliação de Expressões

```
def eval_expression(expr, local_context=None):
if local_context is None:
local_context = names
if isinstance(expr, tuple):
if expr[0] == 'assign':
local_context[expr[1]] = eval_expression(expr[2], local_context)
elif expr[0] == 'write':
return eval_expression(expr[1], local_context)
elif expr[0] == 'read':
return local_context[expr[1]]
elif expr[0] == 'random':
return local_context[expr[1]]
elif expr[0] == 'function_def':
functions[expr[1]] = (expr[2], expr[3])
elif expr[0] == 'greater_than':
left = eval_expression(expr[1], local_context)
right = eval_expression(expr[2], local_context)
return left > right
elif expr[0] == 'less_than':
left = eval_expression(expr[1], local_context)
right = eval_expression(expr[2], local_context)
return left < right
elif expr[0] == 'function_call':
func_name = expr[1]
args = [eval_expression(arg, local_context) for arg in expr[2]]
param_names, body = functions[func_name]
saved_context = local_context.copy()
local_context = {}
for param, arg in zip(param_names, args):
local_context[param] = arg
result = None
if isinstance(body, list):
for sub_stmt in body:
result = eval_statement(sub_stmt, local_context)
else:
result = eval_statement(body, local_context)
local_context.clear()
local_context.update(saved_context)
return result
elif isinstance(expr, int):
return expr
elif isinstance(expr, str):
return local_context.get(expr, expr)
elif isinstance(expr, list):
return [eval_expression(e, local_context) for e in expr]
elif isinstance(expr, str):
return interpolate_string(expr)
```

A função **eval_expression** é responsável por avaliar diferentes tipos de expressões, incluindo atribuições, operações de leitura e escrita, chamadas de função, comparações e avaliação de listas. Dependendo da estrutura da expressão, a função realiza a operação apropriada e retorna o resultado, utilizando um contexto local para armazenar variáveis e funções temporariamente durante a avaliação.

Regras Auxiliares

Parâmetros

```
def p_parameters(p):
'''parameters : parameters COMMA IDENTIFIER
| IDENTIFIER
| empty'''
if len(p) == 4:
p[0] = p[1] + [p[3]]
elif len(p) == 2 and p[1] is not None:
p[0] = [p[1]]
else:
p[0] = []
```

A função **p_parameters** define como os parâmetros de uma função ou procedimento são reconhecidos e processados pelo analisador sintático. Os parâmetros podem ser uma lista de identificadores separados por vírgulas, um único identificador ou nenhum parâmetro. A função constrói e retorna uma lista de identificadores que representa os parâmetros.

Argumentos

```
def p_arguments(p):
'''arguments : arguments COMMA expression
| expression
| empty'''
if len(p) == 4:
p[0] = p[1] + [p[3]]
elif len(p) == 2 and p[1] is not None:
p[0] = [p[1]]
else:
p[0] = []
```

A função **p_arguments** define como os argumentos de uma função ou procedimento são reconhecidos e processados pelo analisador sintático. Os argumentos podem ser uma lista de expressões separadas por vírgulas, uma única expressão ou nenhum argumento. A função constrói e retorna uma lista de expressões que representa os argumentos.

Operações Binárias

```
def p_expression_binop(p):
'''expression : expression PLUS expression
| expression MINUS expression
| expression TIMES expression
| expression DIVIDE expression
| expression CONCAT expression
| expression MAIOR expression
| expression MENOR expression'''
if p[2] == '+':
p[0] = eval_expression(p[1]) + eval_expression(p[3])
elif p[2] == '-':
p[0] = eval_expression(p[1]) - eval_expression(p[3])
elif p[2] == '*':
p[0] = eval_expression(p[1]) * eval_expression(p[3])
elif p[2] == '/':
p[0] = eval_expression(p[1]) // eval_expression(p[3])
elif p[2] == '<>':
p[0] = str(eval_expression(p[1])) + str(eval_expression(p[3]))
elif p[2] == '/\\':
p[0] = ('greater_than', p[1], p[3])
elif p[2] == '\\/':
p[0] = ('less_than', p[1], p[3])
```

A função **p_expression_binop** define como o analisador sintático deve processar e avaliar expressões binárias, como somas, subtrações, multiplicações, divisões, concatenações e comparações. Dependendo do operador entre as duas expressões, a função avalia e retorna o resultado apropriado, permitindo ao analisador léxico interpretar e manipular corretamente essas operações na linguagem de programação.

Agrupamento de Expressões

```
def p_expression_group(p):
'expression : LPAREN expression RPAREN'
p[0] = p[2]
```

A função **p_expression_group** define como o analisador sintático deve processar expressões entre parênteses. Ela assegura que o valor da expressão dentro dos parênteses seja corretamente avaliado e atribuído, ignorando os próprios parênteses na avaliação final. Isso permite que expressões agrupadas sejam tratadas corretamente na linguagem de programação.

Números e Identificadores

```
def p_expression_number(p):
'expression : NUMBER'
p[0] = p[1]
```

A função **p_expression_numbe**r define como o analisador sintático deve processar números. Quando encontra um número, ele o reconhece como uma expressão válida e atribui esse valor diretamente, permitindo que os números sejam corretamente utilizados e avaliados na linguagem de programação.

```
def p_expression_identifier(p):
'expression : IDENTIFIER'
p[0] = names.get(p[1], p[1])
```

A função **p_expression_identifier** define como o analisador sintático deve processar identificadores. Quando encontra um identificador, ele verifica se existe um valor associado a ele no dicionário **names**. Se existir, usa esse valor, caso contrário, usa o identificador como está. Isso permite que variáveis sejam corretamente reconhecidas e avaliadas na linguagem de programação.

Strings

```
def p_expression_string(p):
'expression : STRING'
p[0] = interpolate_string(p[1])
```

Esta função define como o analisador sintático deve processar strings. Quando encontra uma string, a função **interpolate_string** é utilizada para processar qualquer interpolação dentro da string. O resultado final é então utilizado como a expressão. Isso permite que strings com variáveis ou expressões interpoladas sejam corretamente reconhecidas e avaliadas na linguagem de programação.

Listas

```
def p_expression_list(p):
'expression : LBRACKET elements RBRACKET'
if not p[2]:
p[0] = []
else:
p[0] = p[2]
```

A função **p_expression_list** define como o analisador sintático deve processar listas. Quando encontra uma lista delimitada por parenteses retos, verifica se a lista contém elementos. Se estiver vazia, retorna uma lista vazia (“[none]”). Caso contrário retorna a lista de elementos. Isso permite que listas sejam corretamente reconhecidas e avaliadas na linguagem de programação.

```
def p_elements(p):
'''elements : elements COMMA expression
| expression
| empty'''
if len(p) == 2:
p[0] = [p[1]]
elif len(p) == 4:
p[0] = p[1] + [p[3]]
else:
p[0] = []
```

A função **p_elements** define como o analisador sintático deve processar os elementos de uma lista. Os elementos podem ser uma lista de expressões separadas por vírgulas, uma única expressão ou nenhum elemento. A função constrói e retorna uma lista de expressões que representa os elementos da lista, permitindo que listas sejam corretamente reconhecidas e avaliadas na linguagem de programação.

Expressão Vazia

```
def p_empty(p):
'empty :'
```

A função **p_empty** define uma produção vazia na gramática do analisador sintático. Serve para reconhecer e processar situações onde uma lista de elementos ou outras construções gramaticais podem ser opcionalmente vazias, garantindo que a sintaxe do código seja interpretada corretamente mesmo quando certos elementos estão ausentes.

Tratamento de Erros

```
def p_error(p):
if p:
print(f"Erro de sintaxe em '{p.value}'")
else:
print("Erro de sintaxe no EOF")
```

A função **p_error** define como o analisador sintático deve tratar e reportar erros de sintaxe. Quando ocorre um erro, ela verifica se há um token disponível e imprime uma mensagem indicando onde o erro ocorreu. Se o erro ocorrer no fim do arquivo, uma mensagem específica é exibida. Isso ajuda a identificar e corrigir problemas de sintaxe no código.

Construção do Parser

```
# Constrói o parser
parser = yacc.yacc()

def parse_file(filename):
with open(filename, 'r', encoding='utf-8') as file:
data = file.read()
result = parser.parse(data)
if result:
for stmt in result:
eval_statement(stmt)
```

Este trecho de código constrói o parser e define a função **parse_file**. A função **parse_file** abre um arquivo, lê seu conteúdo, analisa-o com o parser e avalia cada declaração encontrada. O conteúdo do arquivo é lido e armazenado na variável data. Em seguida, o parser é utilizado para analisar esses dados, produzindo um resultado. Se o resultado não for nulo, o código itera sobre as declarações contidas nele, avaliando cada uma usando a função **eval_statement**.

Descrição main.py

```python
import sys
from grammar import parse_file

def main():
    # Verifica se o número de argumentos da linha de comando é igual a 2
    if len(sys.argv) != 2:
        # Imprime a mensagem de uso correto do script
        print("Usage: python main.py <filename>")
        # Encerra o programa com o código de status 1 (indicando um erro)
        sys.exit(1)

    # Obtém o nome do arquivo a partir dos argumentos da linha de comando
    filename = sys.argv[1]
    # Chama a função parse_file com o nome do arquivo
    parse_file(filename)

# Verifica se o script está sendo executado diretamente (e não importado
como módulo)
if __name__ == "__main__":
    # Chama a função principal
    main()
```

Este código define a função **main** para executar o processo de análise de um arquivo especificado via terminal. O script verifica se o número correto de argumentos foi fornecido, obtém o nome do arquivo, e então chama a função **parse_file** para processar o conteúdo do arquivo.

Conclusão

Durante o desenvolvimento deste projeto, focámos na implementação de uma aplicação em Python utilizando a biblioteca PLY para interpretar a Linguagem Funcional do Cávado e do Ave (FCA). Através deste trabalho, adquirimos experiência na definição de analisadores léxicos e sintáticos, bem como na tradução de linguagens por meio de ações semânticas.

Conseguimos resolver totalmente as alíneas A, B e E do enunciado, implementando com sucesso as funcionalidades de atribuição a variáveis, instruções de I/O e strings, e outras instruções básicas. Estes resultados demonstram o nosso entendimento e aplicação das técnicas de análise léxica e sintática, além da construção e avaliação de árvores de sintaxe abstrata.

No entanto, enfrentámos dificuldades significativas nas alíneas C e D. Na alínea C, que trata da declaração de funções, encontrámos desafios ao tentar fazer com que as funções declaradas a partir do input fossem corretamente reconhecidas e executadas. Isso resultou na nossa incapacidade de fazer qualquer exemplo funcionar corretamente nesta secção. Na alínea D, relacionada com listas, conseguimos implementar apenas um exemplo funcional, que foi o exemplo A. Os restantes não atingiram um nível de fiabilidade ou completude satisfatório, uma vez que era necessária a utilização de funções novamente. A nossa maior dificuldade foi a manipulação adequada das funções que vinham do input, refletindo uma necessidade de maior entendimento e prática com estas construções.

Apesar destas dificuldades e da complexidade do trabalho, o desenvolvimento das alíneas concluídas proporcionou-nos uma valiosa experiência e aprendizado na construção de analisadores léxicos e sintáticos e na tradução de linguagens funcionais. Reconhecemos as áreas onde precisamos de melhorar e estamos comprometidos em continuar a desenvolver as nossas habilidades para superar estes desafios em projetos futuros.

Bibliografia

- PLY (Python Lex-Yacc) Documentation. "PLY (Python Lex-Yacc) 4.0 documentation". Disponível em: http://www.dabeaz.com/ply/
- Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2006). Compilers: Principles, Techniques, and Tools (2nd Edition). Pearson.
- Sebesta, R. W. (2016). Concepts of Programming Languages (11th Edition). Pearson.
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd Edition). MIT Press.
- Python Software Foundation. Python Language Reference, version 3.10. Disponível em: https://docs.python.org/3.10/
- "Functional Programming". (2023). In Wikipedia. Disponível em: https://en.wikipedia.org/wiki/Functional_programming