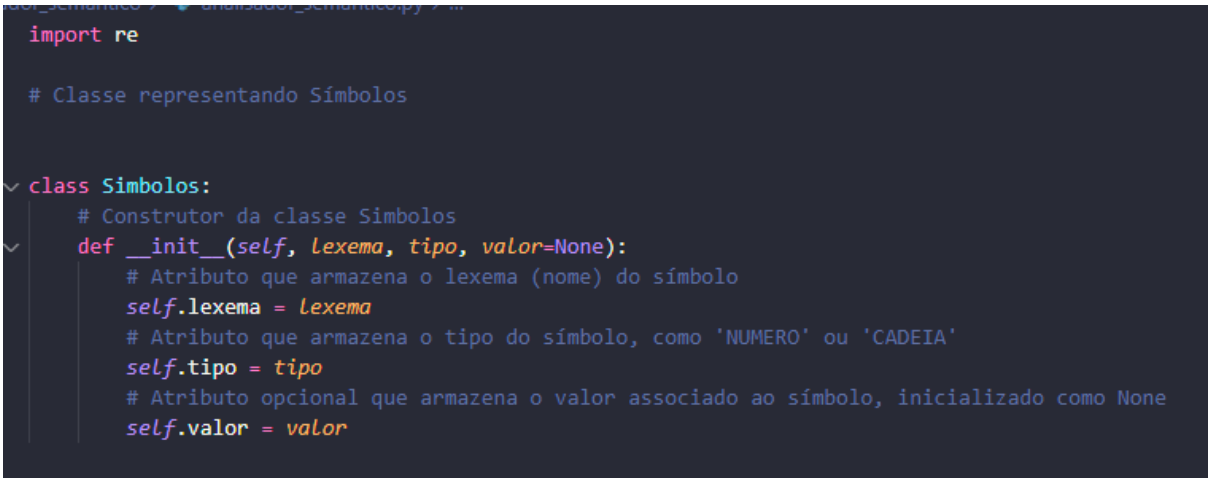
Relatório: Escopo e Verificação de Tipos

Aluno: David Vinícius Pereira Lima
Professora: Jacqueline Midlej do Espírito Santo
Data: 04 de dezembro de 2023

Este trabalho foi implementado em Python e tem como objetivo fazer a análise semântica robusta, gerenciando escopos por meio de tabelas de símbolos, lidando com erros semânticos durante a execução de um linguagem fictícia encontrada no arquivo teste.txt.

Link para o GitHub com o código: https://github.com/Ishinaru/gerenciamento-escopos.git

O Código:



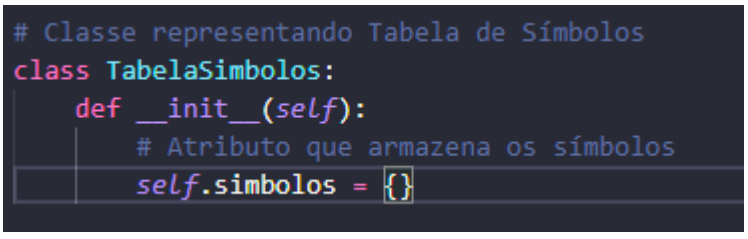
```python
import re

# Classe representando Símbolos


class Simbolos:
    # Construtor da classe Simbolos
    def __init__(self, lexema, tipo, valor=None):
        # Atributo que armazena o lexema (nome) do símbolo
        self.lexema = lexema
        # Atributo que armazena o tipo do símbolo, como 'NUMERO' ou 'CADEIA'
        self.tipo = tipo
        # Atributo opcional que armazena o valor associado ao símbolo, inicializado como None
        self.valor = valor
```

fig1 - classe_simbolos

Começando pela classe Símbolos, que é utilizada para representar os símbolos que podem ser encontrados durante a análise semântica Cada símbolo possui um nome (lexema), um tipo que indica se é um número ou uma cadeia, e um valor opcional associado.



```python
# Classe representando Tabela de Símbolos
class TabelaSimbolos:
    def __init__(self):
        # Atributo que armazena os símbolos
        self.simbolos = {}
```

fig2 - classe_tabela_simbolos

A classe TabelaSimbolos serve como um componente essencial para a implementação de análise semântica em um compilador ou interpretador, mantendo um registro dos símbolos identificados ao longo da execução do programa.

```python
# Classe responsável por realizar análise semântica em um código fictício
class AnalisadorSemantico:
    # Método construtor da classe
    def __init__(self):
        # Atributo que armazena uma lista contendo uma tabela de símbolos inicial
        self.tabela_simbolos = [TabelaSimbolos()]
        # Atributo que define os tipos válidos na linguagem, associando-os a seus tipos correspondentes em Python
        self.tipos_validos = {'NUMERO': (int, float), 'CADEIA': str}

    # Método para executar uma lista de instruções
    def executar_instrucoes(self, instrucoes):
        # Adiciona uma nova tabela de símbolos para representar um novo escopo
        self.tabela_simbolos.append(TabelaSimbolos())
        # Itera sobre as instruções para executar cada uma
        for instrucao in instrucoes:
            self.executar_instrucao(instrucao)
        # Remove a tabela de símbolos após processar todas as instruções do escopo
        self.tabela_simbolos.pop()

    # Método para executar uma instrução
    def executar_instrucao(self, instrucao):
        # Verifica se a instrução é uma lista (pode conter sub-instruções)
        if isinstance(instrucao, list):
            # Se for uma lista, itera sobre as sub-instruções e as executa
            for i in instrucao:
                self.executar_instrucao(i)
        else:
            # Se for uma instrução única
            inst = instrucao["instrucao"]

            # Lógica de execução com base no tipo de instrução
            if inst == "BLOCO":
                # Adiciona uma nova tabela de símbolos para representar um novo escopo (bloco)
                self.tabela_simbolos.append(TabelaSimbolos())
            elif inst == "FIM":
                # Remove a tabela de símbolos para sair do escopo atual (fim do bloco)
                self.tabela_simbolos.pop()
            elif inst == "PRINT":
                # Chama o método para processar a instrução de impressão
                self.processar_print(instrucao["lexema"].strip())
            elif inst in ["ATRIBUICAO", "DECLARACAO"]:
                # Processa instruções de atribuição ou declaração de variáveis
                lexema = instrucao['lexema']
                lexema = re.sub(r'^(NUMERO|CADEIA)\s*', '', lexema).strip()
                instrucao['lexema'] = lexema
                self.add_variavel(instrucao)
            else:
                # Exibe mensagem de erro para instruções inválidas
                print(f"ERRO: Instrução inválida '{inst}")
```

fig3 - classe_analisador_semantico

A classe AnalisadorSemantico fará a análise semântica em do código em si. Essa análise inclui a validação de escopos, execução de instruções de impressão, processamento de atribuições e declarações de variáveis, além de lidar com tipos de dados específicos. A classe mantém uma lista de tabelas de símbolos para gerenciar os escopos durante a análise. O construtor inicia a instância com uma tabela de símbolos inicial e tipos válidos associados. Métodos específicos são implementados para executar instruções individuais, lidar com escopos de blocos (BLOCO e FIM), processar instruções de impressão (PRINT), e adicionar variáveis ao escopo corrente. O código também trata instruções inválidas, exibindo mensagens de erro apropriadas.

```python
def att_valor_variavel(self, lexema, valor):
    # Itera sobre as tabelas de símbolos
    for tabela in self.tabela_simbolos:
        # Verifica se o lexema está presente na tabela atual
        if lexema in tabela.simbolos:
            # Verifica se o valor da variável já foi inicializado
            if tabela.simbolos[lexema].valor is not None:
                tipo_atual = tabela.simbolos[lexema].tipo

                # Compara o tipo atual da variável com o tipo do novo valor
                if tipo_atual == 'CADEIA' and isinstance(valor, str):
                    tabela.simbolos[lexema].valor = valor
                    return
                elif tipo_atual in ['NUMERO', 'CADEIA'] and isinstance(valor, (int, float)):
                    tabela.simbolos[lexema].valor = valor
                    return
                else:
                    # Exibe erro se a atribuição for inválida em termos de tipo
                    print(f"ERRO de Tipo: Atribuição inválida para variável '{lexema}'")
                    return
            else:
                # Exibe erro se a variável não foi declarada
                print(f"ERRO: Variável '{lexema}' não declarada.")
                return

def add_variavel(self, instrucao):
    lexema = instrucao["lexema"].strip()
    tipo_declarado = instrucao.get("tipo_declarado")
    valor = self.processar_valor(instrucao.get("valor"))

    escopo_atual = self.tabela_simbolos[-1]

    # Verifica se a variável já existe no escopo atual
    if lexema in escopo_atual.simbolos:
        # Atualiza o valor se a variável já foi inicializada
        if valor is not None and isinstance(valor, self.tipos_validos[escopo_atual.simbolos[lexema].tipo]):
            escopo_atual.simbolos[lexema].valor = valor
        else:
            # Exibe erro se a atribuição for inválida em termos de tipo
            print(f"ERRO de Tipo: Atribuição inválida para variável '{lexema}'")
    else:
        # Inferência do tipo da variável e criação de um novo símbolo
        tipo_para_usar = tipo_declarado or self.inferir_tipo(valor)
        novo_simbolo = Simbolos(lexema, tipo_para_usar, valor)
        escopo_atual.simbolos[lexema] = novo_simbolo

def processar_valor(self, valor):
    # Retorna o valor formatado removendo as aspas, se necessário
    if valor is None:
        return None
    return valor.strip('"') if valor.startswith('"') and valor.endswith('"') else \
        int(valor) if valor.lstrip('-').isdigit() else \
        float(valor) if '.' in valor or valor.lstrip('-+').replace('.', '').isdigit() else \
        self.get_valor_variavel(valor.strip())

def get_valor_variavel(self, lexema):
    # Obtém o valor da variável percorrendo as tabelas de símbolos
    for tabela in self.tabela_simbolos:
        if lexema in tabela.simbolos and tabela.simbolos[lexema].valor is not None:
            return tabela.simbolos[lexema].valor
    return None

def inferir_tipo(self, valor):
    # Inferência do tipo com base nos tipos válidos definidos
    for nome_tipo, tipo_valido in self.tipos_validos.items():
        if isinstance(valor, tipo_valido):
            return nome_tipo
    return None
```

fig4 - classe_analisador_semantico

Essas funções (fig-4_classe_analisadorSemantico) da classe AnalisadorSemantico tratam da atualização de valores de variáveis, adição de novas variáveis, processamento de valores, obtenção de valores de variáveis e inferência de tipos. Elas garantem a consistência e validade das operações no contexto da análise semântica, exibindo mensagens de erro apropriadas quando necessário.

```python
def processar_print(self, lexema):
    # Obtém o tipo e valor da variável e exibe informações de impressão
    tipo_variavel = self.get_tipo_variavel(lexema)
    valor_variavel = self.get_valor_variavel(lexema)

    # Verifica se o tipo da variável é conhecido
    if tipo_variavel is not None:
        # Exibe informações de impressão
        print(f'PRINT <{lexema}>:\n   Tipo: {tipo_variavel}\nValor: {valor_variavel}')
    else:
        # Exibe erro se a variável não foi declarada
        print(f"ERRO: Variável '{lexema}' não declarada.")

def get_tipo_variavel(self, lexema):
    # Obtém o tipo da variável percorrendo as tabelas de simbolos
    for tabela in self.tabela_simbolos:
        if lexema in tabela.simbolos:
            return tabela.simbolos[lexema].tipo
    return None
```

fig5 - classe_analisador_semantico

Por fim, a função processar_print exibe informações de impressão para a variável especificada (lexema), mostrando o tipo e o valor associado. Se a variável não foi declarada, um erro é exibido.

A função get_tipo_variavel retorna o tipo da variável consultando as tabelas de símbolos. Se a variável não for encontrada em nenhum escopo, retorna None. Essas funções são parte do processo de análise semântica e auxiliam na interpretação e verificação de tipos durante a execução do programa.

```python
class ProcessarSemantica:
    def processar(self, arquivo):
        # Abre o arquivo especificado, lê seu conteúdo e chama o método processar_codigo
        with open(arquivo, 'r', encoding='utf-8') as file:
            conteudo = file.read()
        return self.processar_codigo(conteudo)

    def processar_codigo(self, conteudo):
        # Inicializa uma lista vazia para armazenar as instruções do código
        instrucoes = []

        # Itera sobre as linhas do conteúdo do código
        for linha in conteudo.splitlines():
            # Verifica se a linha não está em branco
            if linha.strip():
                # Chama o método processar_linha para converter a linha em instruções
                instrucao = self.processar_linha(linha)
                # Verifica se instrucao é não nulo antes de adicioná-lo à lista
                if instrucao:
                    instrucoes.append(instrucao)

        # Retorna a lista de instruções processadas
        return instrucoes

    def processar_linha(self, linha):
        # Divide a linha em duas partes, onde a primeira é o tipo de instrução
        partes = linha.split(maxsplit=1)
        tipo_instrucao = partes[0]

        # Verifica o tipo de instrução e chama o método correspondente
        if tipo_instrucao == "BLOCO" or tipo_instrucao == "FIM":
            return {"instrucao": tipo_instrucao, "nome_bloco": partes[1].strip()}
        elif "=" in linha:
            return self.processar_atribuicao(linha)
        elif tipo_instrucao in {"NUMERO", "CADEIA"}:
            return self.processar_declaracao(linha)
        elif tipo_instrucao == "PRINT":
            return {"instrucao": tipo_instrucao, "lexema": partes[1].strip()}

    def processar_atribuicao(self, linha):
        # Divide as atribuições separadas por virgula e cria instruções correspondentes
        declaracoes = linha.split(",")
        instrucoes = []

        for declaracao in declaracoes:
            lexema, valor = [parte.strip() for parte in declaracao.split("=")]
            instrucoes.append({"instrucao": "ATRIBUICAO",
                               "lexema": lexema, "valor": valor})

        # Retorna a lista de instruções de atribuição
        return instrucoes

    def processar_declaracao(self, linha):
        # Divide a linha em duas partes, onde a primeira é o tipo declarado
        partes = linha.split(maxsplit=1)
        tipo_declarado = partes[0]
        # Divide as declarações separadas por virgula e cria instruções correspondentes
        declaracoes = [variavel.strip() for variavel in partes[1].split(",")]
        instrucoes = [{"instrucao": "DECLARACAO", "lexema": variavel,
                       "tipo_declarado": tipo_declarado} for variavel in declaracoes]

        # Retorna a lista de instruções de declaração
        return instrucoes
```

fig6 - classe_processar_semantica

A classe ProcessarSemantica (fig-6_classe_processarSemantica) contém métodos para processar o código-fonte fictício. O método processar recebe o nome de um arquivo, lê seu conteúdo e chama processar_codigo. O método processar_codigo itera sobre as linhas do código, chama processar_linha para cada linha não vazia e acumula as instruções em uma lista.
O método processar_linha divide a linha em partes, identifica o tipo de instrução e chama métodos específicos para processar atribuições, declarações e instruções de impressão. As instruções processadas são retornadas como um dicionário.
Os métodos processar_atribuicao e processar_declaracao dividem as atribuições ou declarações, criam instruções correspondentes e as retornam como listas de dicionários.

```python
def main():
    # Cria uma instância da classe ProcessarSemantica
    processador = ProcessarSemantica()

    # Cria uma instância da classe AnalisadorSemantico
    analisador = AnalisadorSemantico()

    # Especifica o nome do arquivo a ser processado
    arquivo = "teste.txt"

    # Chama o método processar da instância de ProcessarSemantica para obter as instruções do arquivo
    instrucoes = processador.processar(arquivo)

    # Chama o método executar_instrucoes da instância de AnalisadorSemantico para analisar e executar as instruções
    analisador.executar_instrucoes(instrucoes)


if __name__ == "__main__":
    # Executa a função main se o script estiver sendo executado diretamente
    main()
```

fig7 - função_main

A função main é o ponto de entrada principal do script. Dentro dela, são criadas instâncias das classes ProcessarSemantica e AnalisadorSemantico. Nela iremos testar o arquivo texte.txt que foi dado como exemplo pela professora, e chamado o método processar da instância de ProcessarSemantica para obter as instruções do arquivo.

Posteriormente, as instruções são passadas para o método executar_instrucoes da instância de AnalisadorSemantico, que analisa e executa as instruções.

Instruções para executar:
Para executar, basta ter o python3 instalado em sua máquina e de preferência uma IDE, (recomendo o Visual Studio Code) e rodar o arquivo analisador_semantico.py. Como padrão ele irá rodar o exemplo proposto pela professora, mas caso queira testar com outro programa com a mesma sintaxe, basta criar um arquivo txt na pasta do projeto e trocar na função a atribuição do arquivo = “teste.txt” para o arquivo que você criou.

Resultado (teste.txt):

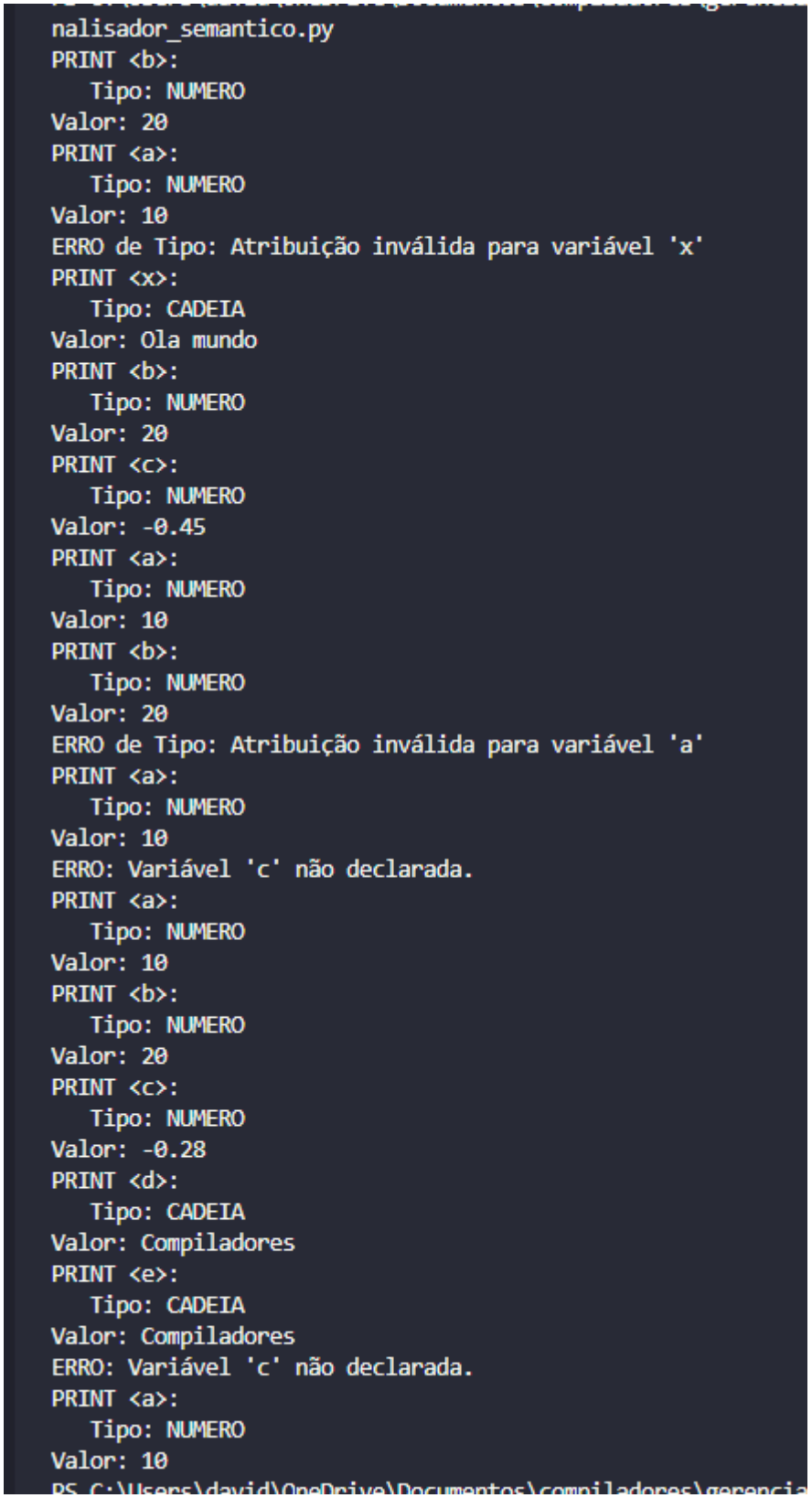


fig8 - saída_teste.txt