Relatório TP2:

1. Classe Tabela

A classe TabelaSimbolos foi desenvolvida com o propósito de gerenciar e armazenar informações sobre identificadores em um contexto de análise léxica e sintática. Este relatório visa descrever as funcionalidades principais da classe e fornecer uma análise crítica de sua implementação.

existeIdent(self, nome): verifica se um identificador já está presente na tabela. Esta função é crucial para evitar a declaração duplicada de identificadores. No entanto, a implementação poderia ser simplificada utilizando diretamente o resultado da verificação (ou seja, return nome in self.tabela).

declaraIdent(self, nomes, valor,linha): é responsável por adicionar identificadores à tabela, juntamente com seus valores associados. Este método realiza a verificação de existência antes de adicionar um identificador, para garantir a integridade da tabela de símbolos.

2. Analisador Sintático

2.1. Mudanças para criação da tabela de simbolos:

A função DECL_TIPO agora cria uma lista para armazenar ids declarados, que são adicionados na função LIST_ID. A função TIPO retornar o tipo dos ids na lista e logo após chamamos a função declaraIdent para salvar ids e tipo no dicionario da tabela.

```
#DECL_TIPO → LIST_ID dpontos TIPO pvirg
def DECL_TIPO(self):
    nomes=[] #lista de para salvar ids
    self.LIST_ID(nomes)
    self.consome(tt.DPONTOS)
    valor =self.TIPO() #salva o tipo
    self.tabsimb.declaraIdent(nomes, valor,self.tokenAtual.linha) #declara os ids na tabela de simbolos
    self.consome(tt.PVIRG)

#LIST_ID → id E
def LIST_ID(self, nomes=[]):
    if self.tokenEsperadoEncontrado(tt.ID):   #se for um id
        nomes.append(self.tokenAtual.lexema)  #adiciona na lista de ids
    self.consome( tt.ID )
    self.E(nomes)
```

2.2. Criação do modo de pânico

A função consome agora entra no modo pânico ao achar um erro, o modo pânico consiste em achar o próximo token de sincronismo para o analisador retornar a fazer as verificações sintáticas.

```
def consome(self, token, NaoTerminal):
    if not self.modoPanico and self.tokenEsperadoEncontrado( token ):
        # tudo seguindo de acordo
        self.tokenAtual = self.lex.getToken()

    elif not self.modoPanico:
        # agora deu erro, solta msg e entra no modo panico
        self.modoPanico = True
        self.deuErro = True
        (const, msg) = token
        print('ERRO DE SINTAXE [linha %d]: era esperado "%s" mas veio "%s"'
           % (self.tokenAtual.linha, msg, self.tokenAtual.lexema))
        #quit()
        procuraTokenDeSincronismo = True
        tokensDeSincronismo = [tt.PVIRG, tt.FIMARQ]
        tokensDeSincronismo.extend(self.follow[NaoTerminal])
        while procuraTokenDeSincronismo:
            self.tokenAtual = self.lex.getToken()
            for tk in tokensDeSincronismo:
                (const, msg) = tk
                if self.tokenAtual.const == const:
                    # tokenAtual eh um token de sincronismo
                    procuraTokenDeSincronismo = False
                    break
    elif self.tokenEsperadoEncontrado(token):
        # chegou no ponto de sincronismo :)
        self.tokenAtual = self.lex.getToken()
        self.modoPanico = False
    else:
        # so continua, consumindo e consumindo...
        pass
```

Os tokens de sincronismo usados foram o ponto e vírgula, o fim de arquivo, e uma lista com os follows da regra. Para isso foi criado um dicionário e a função consome agora deve ser passado com um parâmetro que representa o nome do Não terminal atual.

```
self.follow = {'PROG' : [tt.FIMARQ],
               'DECLS' : [tt.ABRECH],
               'LIST_DECLS' : [tt.ABRECH],
               'D' : [tt.ABRECH],
               'DECL_TIPO' : [tt.ID, tt.ABRECH],
               'LIST_ID' : [tt.DPONTOS, tt.FECHAPAR],
               'E' : [tt.DPONTOS, tt.FECHAPAR],
               'TIPO' : [tt.PVIRG],
               'C_COMP' : [tt.FIMARQ,tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'LISTA_COMANDOS' : [tt.FECHACH],
               'G' : [tt.FECHACH],
               'COMANDOS' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'SE' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'H' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'ENQUANTO' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'LEIA' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'ATRIBUICAO' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'ESCREVA' : [tt.IF,tt.WHILE,tt.READ,tt.WRITE,tt.ID,tt.FECHACH],
               'LIST_W' : [tt.FECHAPAR],
               'L' : [tt.FECHAPAR],
               'ELEM_W' : [tt.VIRG,tt.FECHAPAR],
               'EXPR' : [tt.PVIRG,tt.VIRG,tt.FECHAPAR],
               'P' : [tt.PVIRG,tt.VIRG,tt.FECHAPAR],
               'SIMPLES' : [tt.OPREL,tt.PVIRG,tt.VIRG,tt.FECHAPAR],
               'R' : [tt.OPREL,tt.PVIRG,tt.VIRG,tt.FECHAPAR],
               'TERMO' : [tt.OPAD,tt.OPREL,tt.PVIRG,tt.VIRG,tt.FECHAPAR],
               'S' : [tt.OPAD,tt.OPREL,tt.PVIRG,tt.VIRG,tt.FECHAPAR],
               'FAT' : [tt.OPMUL,tt.OPAD,tt.OPREL,tt.PVIRG,tt.VIRG,tt.FECHAPAR],
              }
```