Analisador Léxico e Sintático em Python

Gabriel Teixeira Júlio e Marcus Vinícius Nogueira Santos

O objetivo deste código é a criação de um simples analisador léxico, este foi feito em Python e se encontra na pasta ***src***. Nela a três arquivos:

- ***main.py*** - arquivo para executar o analisador
- ***analisadorLexico.py*** - arquivo que tem a implementação do AFD, tabela de tokes e tabela de símbolos
- ***tabelaLL1.py*** - arquivo que tem a implementação da tabela da gramática LL(1)
- ***analisadorSintatico.py*** - arquivo que está a implementação do analisador sintático

Implementação

main.py

No arquivo *main.py* inicialmente é requisitado o nome do arquivo da gramática e o símbolo inical, depois é feita a inicialização do AFD, a tabela de símbolos, lista de tokens e do analisador sintático. Depois perguntado o nome do arquivo do código a ser testado, após isso é feita análise léxica, depois é printada a tabela de símbolos e lista de tokens. Por último, é feita a análise sintática do código.

```python
import analisadorLexico as lex  # Importa o módulo do analisador léxico como 'lex'
import analisadorSintatico as sin # Importa o módulo do analisador sintático como 'sin'

gramatica = input("Nome do arquivo da gramática: ")
simbolo_inical = input("Símbolo inical da grmática: ")

# Inicializa o AFD, a tabela de símbolos, a lista de tokens e o analisador sintático
afd = lex.AFD()
simbolos = lex.Simbolos()
tokens = lex.Tokens()
sintatico = sin.Sintatico(gramatica, simbolo_inical)

# Abre o arquivo de código fonte
codigo = input("Nome do arquivo do codigo: ")
file = open(codigo, 'r')
lines = file.readlines()
linha = 0

# Loop para testar cada linha do código fonte
for line in lines:
    linha += 1  # Incrementa o contador de linhas
    resp = afd.testaLinha(simbolos, tokens, linha, line)  # Chama a função para testar a linha
    if resp == False:  # Se houver um erro léxico
        break  # Sai do loop

# Imprime a lista de tokens e a tabela de símbolos
print('Lista de Tokes')
print(tokens)
print('Tabela de Simbolos')
print(simbolos)

# Testa se a lista de tokens está sintaticamente correta
sintatico.testaTokens(tokens)
```

analisadorLexico.py

Como dito anteriormente neste arquivo está a implementação do AFD, tabela de tonkes e tabela de símbolos. O AFD foi feito com base no AFD do arquivo "*AFDCompleto.pdf*" e os tokens foram criados baseados no arquivo "*Exemplos_Linguagem.pdf*.

AFD

O AFD é implementado em uma classe que tem o seguintes atributos:

- **inicial** - o estado inicial do autômato
- **atual** - o estado atual em que o autômato está
- **final** - todos os estados finais, e para cada estado o tipo de Token que ele representa e retorno que serve para dizer se precisa ou não voltar um caracter na cadeia sendo testada. Exemplo: "*'q2': {'tipo': 'ID', 'retorno': -1}*"
- **transicao** - todas as transições do autômato, para cada estado tem se todas suas transições em que cada transição tem os caracteres válidos da transcição e o estado de destino. Exemplo: "*'q5': [[D, 'q6']],*"

O método ***testaCaracter*** testa se caracter passado para o método manda para um estado válido, se vai para estado final ou se vai para um estado ínvalido. Ele segue a seguinte lógica:

1. Começa pegando todas as transições possíveis do estado atual do AFD
2. Para cada transição obtida é feito:
    1. Verifica se o caracter faz parte dos caracteres válidos da transição, senão for parte pula para próxima transição
    2. Se for parte, o estado atual é atualizado para o estado de destino da transição
    3. Verifica se o estado atual é um estado final
        1. Se for final pega as informações do estado final, ou seja, o token
        2. Resta o estado atual para o inicial
        3. Retorna o Token, tipo e se precisa voltar um caracter da cadeia de teste
    4. Senão for o método retorna vazio
3. Se passar por todas as transições e não for caracter de nenhuma retorna um erro

O método ***testaLinha*** que testa uma linha inteira do código sendo analisado pois o AFD só testa um caracter por vez. O método usa alguns variáveis de apoios sendo elas:

- **count** - caracter da linha sendo analisado
- **word** - a palavra que está sendo montada ao passar pela linha, é reniciado quando um token é encontrado
- **resposta** - booleana para retornar se tudo ocorreu como devia ou não

O método segue a seguinte lógica:

1. Enquanto *count* for menor que o tamanho da lina
    1. Soma mais 1 em *count* e pega o caracter da linha na posição *count*
    2. Testa o caracter no AFD usando *testaCaracter*
    3. Atualiza o *count* com o valor de retorno do método *testaCaracter*
    4. Verifica se o método *testaCaracter* retornou um erro
    5. Se tiver retornado insere um Token 'ERRO' na tabela de tokens, *resposta* vira *False* e quebra o loop
    6. Senão retornou um erro:
        1. Adiciona o caracter testado em *word*
        2. Verifica o método *testaCaracter* retornou o tipo *SPACE* ou *BREAKLINE* se tiver reseta *word*
        3. Verica se o retorno de *testaCaracter* é diferente de '', *SPACE* e *BREAKLINE*
        4. Se for verifica se no retorno de *testaCaracter* pede para voltar um caracter, se pedir remove o ultimo caracter de *word*
        5. Verifica se o retorno de *testaCaracter* é *ID*
            1. Se for verifica se é algum Token que é identificado como *ID* usnado *subtiposId* e guarda o resultado em *resp*
            2. Verifica se ainda é *ID* se for verifica se o *ID* já está na tabela de Símblos e guarda o retorno em *resp*
        6. Senão for e verifica se o retorno de *testaCaracter* for >, >=, <, <=, *!*= ou == se for *resp* vira o Token de *COMP*
        7. Senão *resp* recebe o tipo de Token do retorno de *testaCaracter*
        8. Verifica se *resp* é um Token *COMMENT*
            1. Se for reseta *word* pula para próxima iteração do loop
        9. Inseri o Token *resp* na tabela de Tokens
        10. Reseta *word*
2. Retorna *resposta*

```python
# Função para testar uma linha do código fonte
def testaLinha(self, simbolos, tokens, linha, line):
    count = -1  # Inicializa um contador para os caracteres na linha
    word = ''  # Inicializa uma string vazia para armazenar os caracteres
    resposta = True  # Inicializa a variável de resposta como True

    while count < len(line)-1:  # Loop enquanto o contador for menor que o comprimento da linha
        count += 1  # Incrementa o contador
        caracter = line[count]  # Obtém o caracter na posição do contador

        retorno = self.testaCaracter(caracter)  # Chama o método testaCaracter do autômato finito determinístico (AFD) para testar o caracter
        count += retorno[1]  # Incrementa o contador com o retorno do método testaCaracter

        # Verifica se ocorreu um erro léxico ou um erro
        if retorno[0] == 'erro' or retorno[0] == 'ERRO':
            tokens.inserirToken(('ERRO', 'Tem um erro lexico na linha {}'.format(linha)))  # Insere um token de erro na lista de tokens
            resposta = False  # Define a resposta como False
            break  # Sai do loop

        else:
            word += caracter  # Adiciona o caracter à palavra em construção

            if retorno[0] == 'SPACE' or retorno[0] == 'BREAKLINE':  # Se o retorno for um espaço em branco ou quebra de linha
                word = ''  # Reinicia a palavra

            if retorno[0] != '' and retorno[0] != 'SPACE' and retorno[0] != 'BREAKLINE':  # Se o retorno não for vazio, espaço ou quebra de linha
                if retorno[1] == -1:  # Se o retorno for -1, remove o último caracter da palavra
                    word = word[:-1]

                if retorno[0] == 'ID':  # Se o retorno for um identificador
                    resp = subtiposId(retorno[0], word)  # Obtém o subtipo do identificador

                    if resp[0] == 'ID':  # Se o subtipo for ID
                        resp = simbolos.findID(resp)  # Verifica se o identificador já existe na tabela de símbolos

                elif retorno[0] == '>' or retorno[0] == '>=' or retorno[0] == '<' or retorno[0] == '<=' or retorno[0] == '!=' or retorno[0] == '==':
                    resp = ('COMP', retorno[0])  # Se o retorno for um operador de comparação, define o token como 'COMP'

                else:
                    resp = (retorno[0], word)  # Caso contrário, define o tipo como o próprio retorno e a palavra

                if resp[0] == 'COMENT':  # Se o tipo for um comentário
                    word = ''  # Reinicia a palavra
                    continue  # Pula para a próxima iteração do loop

                tokens.inserirToken(resp)  # Insere o token na lista de tokens
                word = ''  # Reinicia a palavra
    return resposta  # Retorna a resposta
```

```python
class AFD:
    def __init__(self):
        # Definição dos estados inicial e atual, e estados finais com tipos e retornos associados
        self.inicial = 'q0'
        self.atual = 'q0'
        self.final = {…
        # Transições de estado para cada estado atual
        self.transicao = {…

    # Método para testar cada caracter e transitar para o próximo estado
    def testaCaracter(self, caracter):
        trns = self.transicao[self.atual]  # Transições possíveis do estado atual

        for i in trns:
            if re.fullmatch(i[0], caracter) != None:  # Verifica se o caractere corresponde à transição
                self.atual = i[1]  # Atualiza o estado atual
                if self.atual in self.final:  # Se o estado atual for final
                    temp = self.final[self.atual]  # Obtém as informações associadas ao estado final
                    self.atual = self.inicial  # Reinicia o estado atual
                    return (temp['tipo'], temp['retorno'])  # Retorna o tipo e o retorno do token
                else:
                    return ('', 0)  # Caso contrário, retorna uma tupla vazia
        else:
            return ('erro', 0)  # Se não houver transição correspondente, retorna um erro
```

Símbolos

A tabela de Símbolos é uma lista que guarda apenas os IDs idetificados pelo analiador.

O método ***inserirID*** serve armazenar na lista de símbolos o valor do ID e retorna para tabela de tokens o token deste ID, enviando como tipo ID e valor o índice do ID inserido na lista de símbolos.

O método ***findID*** é utilizado para verificar se um ID já está na lista de símbolos. Se ele estiver na lista de símbolos ele retorna para tabela de tokens o token deste ID, enviando como tipo ID e valor o índice do ID inserido na lista de símbolos, senão ele utiliza o método *inserirID* para inserir o ID na tabela de símbolos.

```python
# Classe para armazenar os identificadores encontrados
class Simbolos:
    def __init__(self):
        self.ids = []

    def __str__(self):
        return tabulate({'ID': self.ids}, headers=['INDEX', 'ID'], tablefmt="outline", showindex='always')

    # Método para inserir um identificador na lista de identificadores
    def inserirID(self, token):
        # Adiciona o identificador à lista
        self.ids.append(token[1])
        # Retorna o tipo e o índice do identificador na lista
        return ('ID', self.ids.index(token[1]))

    # Método para encontrar um identificador na lista de identificadores
    def findID(self, token):
        # Verifica se o identificador já está na lista
        if token[1] in self.ids:
            # Retorna o tipo e o índice do identificador na lista
            return ('ID', self.ids.index(token[1]))
        else:
            # Caso contrário, insere o identificador na lista e retorna suas informações
            return self.inserirID(token)
```

Tokens

A tabela de Tokens é uma lista em que em cada posição do lista é guardado dois valores:

- **Token** - guardo qual o tipo de token encontrado
- **Value** - guarda o valor do token para os tonkes que necessitam, exemplo para toknes ID armazena o índice do ID na tabela de símbolos

O método ***inserirID*** serve para inserir um token(tipo, valor) na lista de tokens.

```python
# Classe para armazenar os tokens encontrados
class Tokens:
    def __init__(self):
        self.tokens = []

    def __str__(self):
        return tabulate(self.tokens, headers=['Token', 'Value'], tablefmt="outline")

    # Método para inserir um token na lista de tokens
    def inserirToken(self, token):
        self.tokens.append(token)  # Adiciona o token à lista
```

subtiposId

Para alguns IDs é necessário verificar se ele não é um Token válido, como INT ou FLOAT, pois o AFD não separa estes tokens de ID. Então o método ***subtiposId*** foi criado para verificar se um ID é um Token especial ou apenas um ID, por fim ele retorna o token correto.

```
# Função para retornar o subtipo do identificador
def subtiposId(token, word):
    tipos = ['int', 'float', 'char', 'boolean', 'void', 'if', 'else', 'for', 'while', 'scanf', 'println', 'main', 'return']

    if word in tipos:  # Verifica se o identificador é uma palavra-chave
        return (word.upper(), '')  # Retorna o token
    else:
        return ('ID', word)  # Caso contrário, retorna o tipo como ID e o próprio identificador
```

tabelaLL1.py

Como dito anteriormente neste arquivo está a implementação da tabela da gramática LL1. Para armazenar tabela foi criada a classe ***TableLL1*** que possui os atributos:

- ***start_symbol*** - símbolo inicial da gramática
- ***nts*** - guarda todos os não terminais da gramática
- ***ts*** - guarda todos os terminais da gramática tirado 'ε'
- ***parsing_table*** - que guarda a tabela de análise da LL(1)

A classe possui os seguintes métodos:

- ***LL1PeloArquivo*** - método para ler a gramática LL(1) de um arquivo e retorna ela num dicionário
- ***NaoTerminais*** - retorna todos os não terminais da gramática
- ***Terminais*** - retorna todos os terminais da gramática
- ***ConjuntosFirst*** - retorna todos os conjuntos First da grmática
- ***ConjuntosFollow*** retorna todos os conjuntos Follow da gramática
- ***GerarTabelaLL1*** - retorna a tabela de análise da gramática
- ***RetornarDerivacao*** - retorna a derivação de um não terminal para o terminal

No método contrutor da classe é passado o nome do arquivo da gramática e o símbolo inicial, depois é guardado o símbolo inicial, depois é pega a gramática pelo arquivo passado, após é armazendo os não terminais e os terminais, depois são gerados os conjuntos First e Follow e por último é gerada a tabela de análise.

```python
# Classe para armazenar a tabela de análise LL(1)
class TableLL1:
    def __init__(self, filename, start_symbol):
        # Guarda do símbolo inicial
        self.start_symbol = start_symbol

        # Pega a gramática LL(1) pelo arquivo passado
        grammar = self.LL1PeloArquivo(filename)

        # Guarda todos os não terminais e terminais da grámatica
        self.nts, self.ts = self.NaoTerminais(grammar), self.Terminais(grammar)

        # Gera os conjuntos First da gramática
        first_sets = self.ConjuntosFirst(grammar)

        # Gera os conjuntos Follow da gramática
        follow_sets = self.ConjuntosFollow(grammar, start_symbol, first_sets)

        # Gera e armazena a tabela de análise
        self.parsing_table = self.GerarTabelaLL1(grammar, first_sets, follow_sets)

    # Carrega a gramática a partir de um arquivo, retornando um dicionário onde as chaves
    # são os não terminais e os valores são listas de produções.
    def LL1PeloArquivo(self, filename):···

    # Retorna um conjunto com todos os símbolos não terminais
    def NaoTerminais(self, grammar):···

    # Retorna um conjunto com todos os símbolos terminais
    def Terminais(self, grammar):···

    # Metódo para gerar todos os conjuntos First
    def ConjuntosFirst(self, grammar):···

    # Metódo para gerar todos os conjuntos FOllow
    def ConjuntosFollow(self, grammar, start_symbol, first_sets):···

    # Metódo para gerar a tabela de análise
    def GerarTabelaLL1(self, grammar, first_sets, follow_sets):···

    # Função para retorna a derivação de um símblo para um símbolo da entrada
    def RetornarDerivacao(self, symbol, input):···
```

LL1PeloArquivo

O método segue a seguinte lógica:

1. Inicialização da variável que guarda a gramática
2. Abre o arquivo para leitura pelo nome passado
3. Para cada linha no arquivo
    1. Se a linha está vazia pula ela
    2. Se não tiver '->' na linha pula ela
    3. Separa a linha por '->' e guarda o lado esquerdo (lhs) e lado direito (rhs)
    4. Separa as produções no lado direito (rhs) usando o '|' como separador

5. Para cada produção
    1. Verifica se há dois '|' seguidos, indicando um terminal '||'
        1. Adiciona '||' ao terminal
    6. As produções são adicionadas à lista associada ao lado esquerdo (lhs) no dicionário
4. Retorna a gramática como um dicionário

```
# Carrega a gramática a partir de um arquivo, retornando um dicionário onde as chaves
# são os não terminais e os valores são listas de produções.
def LL1PeloArquivo(self, filename):
    grammar = defaultdict(list)
    try:
        # Abre o arquivo da gramática
        with open(filename, 'r', encoding='utf-8') as file:
            # Lê o o arquivo linha por linha
            for line in file:
                line = line.strip()
                # Se não tiver nada na linha pula
                if not line:
                    continue
                # Se não tiver -> na linha, a linha é ínvalida pois todas as linhas precisam ter uma derivação
                if '->' not in line:
                    print(f"Skipping invalid line: {line}")
                    continue
                # lhs recebe o não terminal e rhs recebe as produções de lhs
                lhs, rhs = line.split('->', 1)
                lhs = lhs.strip()
                # Separa as produções de rhs
                productions = [prod.strip() for prod in rhs.split('|')]
                # EXPR_OU2 -> || EXPR_E EXPR_OU2 | ε
                # Como está produção usa dois | (||) como terminal temos que tratar está produção para ela funcionar
                for prod in range(len(productions)):
                    if productions[prod] == "" and productions[prod+1] == "":
                        productions = productions[2:]
                        productions[0] = '|| ' + productions[0]
                        break
                # Adiona a chave lhs e as produções na gramática
                grammar[lhs].extend(productions)
    except FileNotFoundError:
        print(f"Erro: O arquivo {filename} não foi encontrado.")
        return {}
    except IOError as e:
        print(f"Erro ao ler o arquivo {filename}: {e}")
        return {}

    # Retorna a gramática no arquivo
    return dict(grammar)
```

NaoTerminais

O metódo retorna uma lista com todos os não terminais de uma gramática, ou seja, todas as chaves do dicionário.

```
# Retorna um conjunto com todos os símbolos não terminais
def NaoTerminais(self, grammar):
    return set(grammar.keys())
```

Terminais

O método segue a seguinte lógica:

1. Pega todos os não terminais da gramática

2. Inicializa um conjunto vazio para guardr os terminais
3. Para cada lista de produção na gramática
    1. Para cada produção na lista
        1. Separa os símbolos da produção
        2. Para cada símbolo da produção
            1. Verifica se o símbolo não é ε e se o símbolo não é um terminal
                1. Adiciona o símbolo no conjunto de terminais
4. Retorna o conjunto de terminais

```
# Retorna um conjunto com todos os símbolos terminais
def Terminais(self, grammar):
    # Pega todos os não terminais
    non_terminals = self.NaoTerminais(grammar)
    terminals = set()
    # Para cada lista de produções na gramática
    for rhs_list in grammar.values():
        # Para cada produção na lista
        for production in rhs_list:
            # Separa os símbolos
            symbols = production.split()
            # Para cada símbolo
            for symbol in symbols:
                # Se símbolo for dirente de ε e não é um não terminal
                if symbol != 'ε' and symbol not in non_terminals:
                    # Adiciona no conjunto de terminais
                    terminals.add(symbol)
    return terminals
```

ConjuntosFirst

O método segue a seguinte lógica:

1. Inicializa para todos os não terminais os conjuntos First vazio
2. Para cada não terminal
    1. Usa o metódo recursivo *EncontrarFirstSet* para encontrar o conjunto First do não terminal
        1. Se o conjunto First já foi calculado
            1. Retorna o cojunto First
        2. Para cada produção do não terminal
            1. Separa os símbolos da produção
            2. Inicializa uma variável auxiliar *all_empty* como True
            3. Para cada símbolo da produção
                1. Se o símbolo for ε
                    1. Adiciona o ε ao cojunto First do não terminal
                    2. *all_empty* recebe False
                    3. Sai do loop
                2. Verifica se o símbolo é um terminal
                    1. Adiciona o símbolo ao cojunto First do não terminal
                    2. *all_empty* recebe False
                    3. Sai do loop

3. Senão
    1. Gera o conjunto First do símbolo
    2. Adiciona o conjunto First do símbolo - {ε} ao conjunto First do não terminal
    3. Se ε não estiver nos símbolos do conjunto First do símbolo
        1. *all_empty* recebe False
        2. Sai do loop
4. Verifica se *all_empty* for True
    1. Adiciona o ε ao conjunto First do símbolo

3. Retorna todos os conjuntos First

```python
# Metódo para gerar todos os conjuntos First
def ConjuntosFirst(self, grammar):
    # Gera todos os conjuntos FIRST como vazios para todos não terminais da gramática
    first_sets = {non_terminal: set() for non_terminal in self.nts}

    # Função recursiva interna para fazer a busca recursiva do First de um símbolo
    def EncontrarFirstSet(symbol):
        # Se o conjunto First de symbol já existe retorna ele
        if first_sets[symbol]:
            return first_sets[symbol]

        # Para cada produção de symbol na grámatica faça
        for production in grammar[symbol]:
            # Pega todos símbolos da produção
            p_symbols = production.split()
            # Variável auxiliar para saber se um não terminal possui ε apenas se todos os seus símbolos produzem ε
            all_empty = True

            # Para cada símbolo nos símbolos da produção
            for p_symbol in p_symbols:
                # Se o símbolo for ε, adiciona ε ao First de symbol e sai do for
                if p_symbol == 'ε':
                    first_sets[symbol].add('ε')
                    all_empty = False
                    break
                # Se o símbolo é um terminal, adiciona o símbolo ao First de symbol e sai do for
                elif p_symbol in self.ts:
                    first_sets[symbol].add(p_symbol)
                    all_empty = False
                    break
                # Se o símbolo é um não terminal
                else:
                    # Encontra o First do símbolo
                    first_p_symbol = EncontrarFirstSet(p_symbol)
                    # Adiciona ao First do symbol o First do símbolo tirando o ε
                    first_sets[symbol].update(first_p_symbol - {'ε'})
                    # Se o ε não estiver no First do símbolo sai do for, senão continua o for
                    if 'ε' not in first_p_symbol:
                        all_empty = False
                        break
            # Se todos os símbolos da produção produzem ε adiciona ε ao First de symbol
            if all_empty:
                first_sets[symbol].add('ε')
        # Retorna o First de symbol
        return first_sets[symbol]

    # Para todos os não terminais gera os conjuntos First
    for non_terminal in self.nts:
        EncontrarFirstSet(non_terminal)

    # Retorna todos os conjuntos First
    return first_sets
```

ConjuntosFollow

O método segue a seguinte lógica:

1. Inicializa para todos os não terminais os conjuntos Follow vazio
2. Adiciona $ ao conjunto Follow do símbolo inicial
3. Para cada não terminal
    1. Usa o metódo recursivo *follow_of* para encontrar o conjunto Follow do não terminal
        1. Se não existe o conjunto Follow para o symbol
            1. Gera o conjunto Follow para o symbol
        2. Para cada não terminal na gramática
            1. Para cada produção do não terminal
                1. Separa os símbolos da produção
                2. Verifica se o symbol aparece na produção
                    1. Pega a posição do símbolo a direita de *symbol*
                    2. Enquanto tiver símbolos a direita
                        1. Pega o próximo símbolo
                        2. Se o próximo símbolo for um não terminal
                            1. Adiciona o conjunto First do próximo símbolo ao conjunto Follow do *symbol*
                            2. Se não tiver ε no conjunto First do próximo símbolo
                                1. Sai do loop
                        3. Senão
                            1. Adiciona o próximo símbolo ao conjunto Follow do *symbol*
                            2. Sai do loop
                    3. Se o próximo símbolo for o último
                        1. Se *symbol* não for o não terminal
                            1. Adiciona o conjunto Follow do não terminal ao conjunto Follow do *symbol*
        3. Retorna o conjunto Follow do *symbol*
4. Retorna todos os conjuntos Follow

```python
# Metódo para gerar todos os conjuntos FOllow
def ConjuntosFollow(self, grammar, start_symbol, first_sets):
    # Inicializa para todos não teminais o conjunto Follow como vazio
    follow = defaultdict(set)
    # Adiciona o $ para o conjunto Follow do símbolo inicial
    follow[start_symbol].add('$')
    # Função recursiva interna para fazer a busca recursiva do Follow de um símbolo
    def follow_of(symbol):
        # Certifica que o símbolo tem um conjunto Follow
        if symbol not in follow:
            follow[symbol] = set()
        # Para cada não terminal na gramática
        for nt in self.nts:
            # Para cada produção do não terminal
            for production in grammar[nt]:
                # Separa os símbolos da produção
                symbols = production.split()
                # Conferi se o símbolo atual está na produção do não terminal
                if symbol in symbols:
                    # Começa a verificar os símbolos após o símbolo atual
                    follow_pos = symbols.index(symbol) + 1
                    # Enquanto o símbolo checado não depois  do último
                    while follow_pos < len(symbols):
                        # Guarda o próximo símbolo
                        next_symbol = symbols[follow_pos]
                        # Se o próximo símbolo for um não terminal
                        if next_symbol in self.nts:
                            # Adiciona o First do próximo símbolo no Follow do símbolo atual, excluindo o ε
                            follow[symbol] |= first_sets[next_symbol] - {'ε'}
                            # Se o First do próximo símbolo não contém o ε, sai do loop
                            if 'ε' not in first_sets[next_symbol]:
                                break
                        # Se o próximo símbolo é um terminal
                        else:
                            # Adiciona o terminal ao Follow do símbolo atual
                            follow[symbol].add(next_symbol)
                            # Sai do loop
                            break
                        # Vai para o próximo símbolo da produção
                        follow_pos += 1
                    # Se chegou ao final da produção ou todos os símbolos restantes derivão em ε
                    if follow_pos == len(symbols):
                        # Evita dependências própias (não adicionar o conjunto Follow do símbolo atual em si mesmo)
                        if nt != symbol:
                            # Adiciona o Follow do não terminal ao Follow do símbolo atual
                            follow[symbol] |= follow_of(nt)
        # Retorna o conjunto Follow do símbolo atual
        return follow[symbol]
    # Para todos os não terminais gera os conjuntos Follow
    for non_terminal in grammar:
        follow_of(non_terminal)
    # Retorna todos os conjuntos Follow
    return dict(follow)
```

GerarTabelaLL1

O método segue a seguinte lógica:

1. Pega todos os terminais mais o '$'
2. Pega todos os não terminais
3. Gera um dicionário para tabela. Cada não terminal (nt) tem um sub-dicionário onde cada terminal (t) mapeia para o valor '-', indicando inicialmente que não há produção associada
4. Para cada não terminal na gramática
    1. Para cada terminal no conjunto First do não terminal
        1. Se o terminal é diferente de ε

1. Para cada produção do não terminal
    1. Separa os símbolos da produção
    2. Para cada símbolo da produção
        1. Se o símbolo é diferente de ε
            1. Se símbolo for um terminal
                1. Se o terminal do First for igual ao símbolo
                    1. Adicona a produção na tabela linha do não terminal e coluna do terminal do First
                2. Se o terminal do First está no conjunto First do símbolo
                    1. Adicona a produção na tabela linha do não terminal e coluna do terminal do First
                3. Se ε no conjunto First do símbolo
                    1. Continua o loop
                4. Sai do loop
    2. Senão
        1. Para cada terminal no conjunto Follow do não terminal
            1. Adicona a produção 'ε' na tabela linha do não terminal e coluna do terminal do Follow
5. Retorna a tabela

```python
# Metódo para gerar a tabela de análise
def GerarTabelaLL1(self, grammar, first_sets, follow_sets):
    # Adiciona o marcador de fim de entrada aos terminais
    terminals = self.ts | {'$'}
    non_terminals = self.nts
    # Começa a tabela com todas posições valendo '-'
    parsing_table = {nt: {t: '-' for t in terminals} for nt in non_terminals}

    # Para cada não teminal
    for non_terminal in grammar.keys():
        # Para cada terminal do cojunto First do não terminal
        for terminal_first in first_sets[non_terminal]:
            # Se o terminal for diferente de ε
            if terminal_first != 'ε':
                # Para cada produção do não terminal na gramática
                for production in grammar[non_terminal]:
                    # Separa os símbolos da produção
                    symbols = production.split()
                    # Para cada símbolo
                    for symbol in symbols:
                        # Se o símbolo for diferente de ε
                        if symbol != 'ε':
                            # Se o símbolo for um terminal
                            if symbol in terminals:
                                # Se o símbolo é o terminal do conjunto First sendo analisado
                                if terminal_first == symbol:
                                    # Adiciona a produção na table na posição linha do não terminal e coluna do terminal do conjunto First sendo analisado
                                    parsing_table[non_terminal][terminal_first] = production
                            # Se o símbolo é um não terminal
                            # Verifica se o terminal do conjunto First está no conjunto First do símbolo
                            elif terminal_first in first_sets[symbol]:
                                # Adiciona a produção na table na posição linha do não terminal e coluna do terminal do conjunto First sendo analisado
                                parsing_table[non_terminal][terminal_first] = production
                            # Se o conjunto First do símbolo possui o ε continua o loop dos símbolos
                            elif 'ε' in first_sets[symbol]:
                                continue
                            # Se não possui ε sai do loop de símbolos
                            break
            # Se o terminal for ε
            else:
                # Para cada terminal do conjunto Follow do não terminal
                for terminal in follow_sets[non_terminal]:
                    # Adiciona a produção ε na table na posição linha do não terminal e coluna do terminal do conjunto Follow sendo analisado
                    parsing_table[non_terminal][terminal] = 'ε'
    return parsing_table
```

RetornaDerivacao

O método serve para retornar a derivação de um símbolo (não terminal) para o símbolo da entrada (terminal) da tabela de análise.

```
# Função para retorna a derivação de um símblo para um símbolo da entrada
def RetornarDerivacao(self, symbol, input):
    return self.parsing_table[symbol][input]
```

analisadorSintatico.py

Para trabalhar com o analisador sintático foi criado a classe ***Sintaico*** que tem o atributo ***ll1*** que é uma classe *TableLL1* que é gerada no método construtor da classe. A classe possui só um método chamado ***testaTokens*** que testa se uma lista de tokens está sintaticamente correta.

testaTokens

O método recebe a lista de tokens gerada pelo analisador léxico, onde uma cópia dela é armazenada como a entrada do analisador sintático. Depois é criada uma pilha vazia para guardar os símbolos utilizados pelo analisador.

O analisador começa como um loop infinito que segue a seguinte lógica:

1. Pega o primeiro símbolo da entrada
2. Verifica se a pilha está vazia e o primeiro símbolo é '$'
    1. Printa que o código está sintaticamente correto.
3. Senão
    1. Pega o topo da pilha
    2. Verifica se o símbolo for um terminal e for diferente do símbolo incial da entrada
        1. Printa que o código está sintaticamente incorreto
        2. Printa que o erro foi ao comparar o topo da pilha com a entrada
    3. Verifica se o símbolo for diferente do símbolo incial da entrada
        1. Pega a derivação do símbolo para o símbolo incial da entrada usando *RetornarDerivacao* e inverte a ordem da derivação
        2. Para cada símbolo da derivação
            1. Se o símbolo for '-'
                1. Printa que o código está sintaticamente incorreto
                2. Printa que o erro foi que não a derivação so símbolo para símbolo incial da entrada
            2. Se o símbolo for diferente de 'ε'
                1. Empilha o símbolo na pilha
    4. Senão
        1. Remove o símbolo inicial da entrada

```python
import tabelaLL1 as ll1

class Sintatico:
    def __init__(self, grammar, start):
        # Inicializa o analisador sintático, carregando a tabela LL(1) da gramática especificada
        self.ll1 = ll1.TableLL1(grammar, start)

    # Metódo principal do analisador léxico, testa se a lista de tokens está sintaicamente correta
    def testaTokens(self, tokens):
        # Cria uma cópia da lista de tokens e adiciona o marcador de fim da entrada ('$')
        input_data = tokens.tokens.copy()
        input_data.append(('$', ''))

        # Inicializa a pilha de símbolos com o símbolo inicial da gramática
        stack = list()
        stack.append(self.ll1.start_symbol)

        # Loop principal do analisador sintático
        while True:
            # Obtém o primeiro símbolo da entrada, convertendo para minúsculas
            first_symbol = input_data[0][0].lower()

             # Verifica se a pilha está vazia e a entrada foi totalmente consumida
            if not stack and first_symbol == '$':
                # Se sim, o código está sintaticamente correto
                print('Código sintaticamente correto!')
                return
            else:
                # Pega o símbolo no topo da pilha
                symbol = stack.pop()
                # Se o topo da pilha é um terminal e não coincide com o símbolo de entrada
                if symbol in self.ll1.ts and symbol != first_symbol:
                    print('Código sintaticamente incorreto')
                    print('Erro ao comparar o topo da pilha com a entrada!')
                    return
                # Se o símbolo não coincide com o primeiro símbolo da entrada
                elif symbol != first_symbol:
                    # Obtém a derivação correspondente na tabela LL(1)
                    derivate = self.ll1.RetornarDerivacao(symbol, first_symbol).split()
                    # Reverte a derivação para inserção correta na pilha
                    derivate = derivate[::-1]

                    # Para cada símbolo na derivação
                    for s in derivate:
                        # Se s for -, não há a derivação na tabela
                        if s == '-':
                            print('Código sintaticamente incorreto')
                            print(f'Não existe derivação para {symbol} com {first_symbol}!')
                            return
                        # Se o símbolo não é ε, empilha ele
                        elif s != 'ε':
                            stack.append(s)
                # Se o símbolo do topo da pilha coincide com o símbolo da entrada, consome o símbolo da entrada
                else:
                    input_data.pop(0)
```

Execução

Para executar o analisador basta executar o arquivo ***main.py*** e passar qual o nome do arquivo da gramática e qual o nome do arquivo do código a ser testado.

Exemplos de entrada e saída

Entrada

Exemplo de entrada correta 1:

```
main () {
    int x;
    x = 2;
    println(x);
}
```

Exemplo de entrada correta 2:

```
int teste (int x) {
    int y;
    y = 10;
}

main () {
    boolean x, y;
    x = 1;
    y = 0;
    while (x) {
        x = y;
    }
}
```

Exemplo de entrada errada:

```
int teste (int x) {
    int y;
    y = 10;
}
```

Saída

As saídas serão a tabela de tokens, a tabela de símbolos e se está sintaticamente correto ou incorreto, além de mostrar qualque foi o erro.

Saída do exemplo de entrada correta 1:

```
Lista de Tokes
+---------+---------+
| Token   | Value   |
+=========+=========+
| MAIN    |         |
| (       | (       |
| )       | )       |
| {       | {       |
```

```
| INT      |          |
| ID       | 0        |
| ;        | ;        |
| ID       | 0        |
| =        | =        |
| NUM_INT  | 2        |
| ;        | ;        |
| PRINTLN  |          |
| (        | (        |
| ID       | 0        |
| )        | )        |
| ;        | ;        |
| }        | }        |
+----------+----------+
Tabela de Simbolos
+----------+------+
|   INDEX  | ID   |
+==========+======+
|        0 | x    |
+----------+------+
Código sintaticamente correto!
```

Saída do exemplo de entrada correta 2:

```
Lista de Tokes
+----------+----------+
| Token    | Value    |
+==========+==========+
| INT      |          |
| ID       | 0        |
| (        | (        |
| INT      |          |
| ID       | 1        |
| )        | )        |
| {        | {        |
| INT      |          |
| ID       | 2        |
| ;        | ;        |
| ID       | 2        |
| =        | =        |
| NUM_INT  | 10       |
| ;        | ;        |
| }        | }        |
| MAIN     |          |
| (        | (        |
| )        | )        |
| {        | {        |
| BOOLEAN  |          |
| ID       | 1        |
| ,        | ,        |
| ID       | 2        |
| ;        | ;        |
```

```
| ID      | 1       |
| =       | =       |
| NUM_INT | 1       |
| ;       | ;       |
| ID      | 2       |
| =       | =       |
| NUM_INT | 0       |
| ;       | ;       |
| WHILE   |         |
| (       | (       |
| ID      | 1       |
| )       | )       |
| {       | {       |
| ID      | 1       |
| =       | =       |
| ID      | 2       |
| ;       | ;       |
| }       | }       |
| }       | }       |
+---------+---------+
Tabela de Simbolos
+---------+-------+
|   INDEX | ID    |
+=========+=======+
|       0 | teste |
|       1 | x     |
|       2 | y     |
+---------+-------+
Código sintaticamente correto!
```

Saída do exemplo de entrada errada:

```
Lista de Tokes
+---------+---------+
| Token   | Value   |
+=========+=========+
| INT     |         |
| ID      | 0       |
| (       | (       |
| INT     |         |
| ID      | 1       |
| )       | )       |
| {       | {       |
| INT     |         |
| ID      | 2       |
| ;       | ;       |
| ID      | 2       |
| =       | =       |
| NUM_INT | 10      |
| ;       | ;       |
| }       | }       |
+---------+---------+
```

```
Tabela de Simbolos
+---------+-------+
|   INDEX | ID    |
+=========+=======+
|       0 | teste |
|       1 | x     |
|       2 | y     |
+---------+-------+
Código sintaticamente incorreto
Não existe derivação para LISTAFUNCOES com $!
```