

Documentação Técnica sobre o Trabalho de Compiladores Análise Léxica e Sintática

Desenvolvedores:

João Paulo de Souza	0035329
Leandro Souza Pinheiro	0015137

Formiga
31 de outubro de 2019

1 - Problema a ser resolvido

O problema a ser resolvido no trabalho é o desenvolvimento das duas primeiras fases do front end ou parte inicial do compilador onde é realizado a correção do código a fim de facilitar sua conversão e execução, este compilador é objetivado para uma linguagem fictícia chamada P, no qual é muito semelhante ao Pascal, entretanto mais simples.

O desenvolvimento consiste em desenvolver o analisador léxico ou scanner, no qual avalia a existência de palavras ou símbolos válidos na linguagem. E também desenvolver o analisador sintático ou parser, no qual avalia a ordem no qual os símbolos e palavras aparecem na linguagem e se eles podem coexistir em determinados contextos.

No fim as duas partes devem ser somadas a fim de detectar erros sintáticos em 45 arquivos de teste cedidos pelo professor.

1.1 Gramática da linguagem:

A -> PROG <EOF>

PROG -> programa id pvirg DECLS C-COMP

DECLS -> | variaveis LIST-DECLS

LIST-DECLS -> DECL-TIPO D

D -> λ | LIST-DECLS

DECL-TIPO -> LIST-ID dpontos TIPO pvirg

LIST-ID -> id E

E -> λ | virg LIST-ID

TIPO -> inteiro | real | logico | caracter

C-COMP -> abrech LISTA-COMANDOS fechach

LISTA-COMANDOS -> COMANDOS G

G -> λ | LISTA-COMANDOS

COMANDOS -> IF | WHILE | READ | WRITE | ATRIB

IF -> se abrepar EXPR fechapar C-COMP H

H -> λ | senao C-COMP

WHILE -> enquanto abrepar EXPR fechapar C-COMP

READ -> leia abrepar LIST-ID fechapar pvirg

ATRIB -> id atrib EXPR pvirg

WRITE -> escreva abrepar LIST-W fechapar pvirg

LIST-W -> ELEM-W L

L -> λ | virg LIST-W

ELEM-W -> EXPR | cadeia

EXPR -> SIMPLES P

P -> λ | oprel SIMPLES

SIMPLES -> TERMO R

R -> λ | opad SIMPLES

TERMO FAT S

S -> λ | opmul TERMO

FAT -> id | cte | abrepar EXPR fecharpar | verdadeiro | falso | opneg FAT}

2 - Estruturas e TAD's

Como o trabalho demanda duas partes que realizam trabalhos diferentes, mas uma depende da outra, decidimos separar as duas partes do compilador em arquivos de desenvolvimento, um para um analisador léxico e outro para o sintático, com seus próprios métodos/funções.

2.1 Scanner.py

Neste arquivo está a classe que realiza a análise léxica, além disso nela também está a instância do arquivo a ser lido. Além disso utilizamos de um dicionário(estrutura da linguagem Python que guarda itens segundo uma chave), para facilitar sua compreensão como token(palavra que pertence a linguagem ou nomear algo que pertence) no processo da análise.

Dado isso, segue abaixo as imagens e breve descrição sobre os métodos desenvolvidos na classe:

- Método para abrir o arquivo a ser lido, que também avalia se o caminho solicitado pelo arquivo existe para ser aberto:

```
# método para abrir o arquivo a ser lido
def open_file(self):
    # verifica se o arquivo já não está aberto
    if (self.file is not None):
        # se estiver, avisa que o arquivo já foi aberto
        return 'File already is open'
    # procura o nome do arquivo no caminho da arquivo
    elif path.exists(self.fileName):
        # caso encontre, abre o arquivo e seta as variaveis
        # que serão utilizados pelo scanner
        self.file = open(self.fileName, 'r', encoding='utf-8')
        self.buffer = ''
        self.line = 1
        # retorna ok
        return 'Ok'
    else:
        # caso entre nesse quer dizer que não encontrou o arquivo
        return 'File not found'
```

- Método para fechar o arquivo a ser lido, que verifica se há arquivo a ser fechado:

```
# método para fechar o arquivo
def close_file(self):
    # se a variavel do arquivo já está vazia, quer dizer que o arquivo já está fechado
    # ou não há arquivo aberto
    if (self.file is None):
        # retorna aviso
        return "There isn't open file"
    else:
        # se houver arquivo aberto, finaliza ele
        self.file.close()
```

- Método que realiza a busca dos caracteres do arquivo um a um, utilizando um buffer para controlar leituras já realizadas:

```
# método para ler os caracteres no arquivo, caractere a caractere
def getChar(self):
    # verifica se o arquivo está aberto
    if self.file is None:
        # se não estiver aberto retorna o erro
        return "There isn't open file"
    # caso o arquivo esteja aberto e o buffer esteja preenchido
    elif len(self.buffer) > 0:
        # pega a primeira posição do buffer
        c = self.buffer[0]
        # remove ele do buffer
        self.buffer = self.buffer[1:]
        # e retorna o caractere lido no buffer
        return c
    else:
        # caso não tenha nada no buffer
        # le um caractere no arquivo
        c = self.file.read(1)
        # se nao foi eof, pelo menos um char foi lido
        # senao len(c) == 0
        if len(c) == 0:
            return None
        else:
            # retorna o caractere em lower case
            return c.lower()
```

- Método acrescentar um caractere que já foi removido novamente ao buffer de leitura, devido a ser necessário lê-lo novamente:

```
# método para buscar um caractere anterior no buffer para que a leitura acompanha a verificação
def ungetChar(self, c):
    # verifica se o caractere não está vazio
    if not c is None:
        # senão estiver, guarda o caractere novamente no buffer
        self.buffer = self.buffer + c
```

- O último método da classe é o método que analisa as sequências de caracteres lidos a fim de encontrar possíveis tokens. Esse método utiliza dos métodos acima para ir buscando caracteres e quando encontra uma sequência que é um token da linguagem, ele retorna este token, que futuramente será inferido seu uso no analisador sintático. Segue abaixo um pequeno trecho de uma leitura de alguns tokens. Este é o método principal da classe, que analisa os tokens e vai classificando-os como os tipos, que foram definidos na especificação da linguagem(consulte a documentação para mais informações) e caso a palavra lida não tenha o padrão de nenhum tipo de token, então será classificado como um token especial para erros.

```

if char == '=':
    return token.Token(self.type.IGUAL, lexem, self.line)
elif char == ';':
    return token.Token(self.type.PVIRG, lexem, self.line)
elif char == ',':
    return token.Token(self.type.VIRG, lexem, self.line)
elif char == '+':
    return token.Token(self.type.OPAD, lexem, self.line)
elif char == '*':
    return token.Token(self.type.OPMUL, lexem, self.line)
elif char == '-':
    return token.Token(self.type.OPAD, lexem, self.line)
elif char == '!':
    return token.Token(self.type.OPNEG, lexem, self.line)
elif char == '(':
    return token.Token(self.type.ABREPAR, lexem, self.line)
elif char == ')':
    return token.Token(self.type.FECHAPAR, lexem, self.line)
elif char == '{':
    return token.Token(self.type.ABRECH, lexem, self.line)
elif char == '}':
    return token.Token(self.type.FECHACH, lexem, self.line)

```

2.2 Parser.py

Neste arquivo está a classe do parser ou analisador sintático, além disso está a instância da tabela de símbolos(estrutura que será explicada mais à frente que guarda os id's que já foram utilizados no programa.) que não será de fato utilizado agora, mas sim na próxima parte do trabalho. Segue abaixo os métodos da classe:

- Método interpretar, que inicia a análise sintática sobre o arquivo a ser lido no analisador léxico, assim o parser, inicia através do método A, que chama o scanner e vai solicitando os tokens para poder realizar a avaliação da sintaxe da linguagem, após o final da interpretação utiliza novamente a classe do scanner para fechar o arquivo. Segue abaixo a imagem do método:

```

#metodo principal da classe, onde iniciara a leitura do arquivo e irá instanciar o scanner
def interpretar(self, nomeArquivo):
    if not self.scanner is None: #caso o scanner ja tenha sido instanciado
        print('ERRO: Já existe um arquivo sendo processado.')#mostra erro
    else:#caso contrario
        #instacia o scanner
        self.scanner = scanner.Scanner(nomeArquivo)
        #abre arquivo a ser compilado
        self.scanner.open_file()
        #pega o primeiro token
        self.current_token = self.scanner.getToken()
        #chama metodo inicial
        self.A()
        #fecha o arquivo compilado corretamente
        self.scanner.close_file()

```

- Método que avalia se o tipo do token atual é igual ao token esperado:


```

#metodo responsavel por realizar a comparacao entre o token atual
# e o token que era esperado
def current_equal_previous(self, token):
    #(const, msg) = token
    #realiza comparacao e retorna TRUE ou FALSE
    return self.current_token.type == token

```

- Método que vai consumir os tokens lidos, ou seja, verifica se o token está como esperado, se sim, pede o próximo e continua a análise:

```

#metodo responsavel por realizar o consumo do token atual
def consume_token(self, type):
    #print(self.current_token.type)
    #verifica se o token atual é igual ao token esperado
    if self.current_equal_previous( type ):
        #se sim pede o novo token
        self.current_token = self.scanner.getToken()
        if (self.current_token.type == self.t.ID):
            self.symbols_table.insert(self.current_token.lexem, self.current_token.type, None, None,
            self.current_token.line)
    else: #caso contrario mostra mensagem de erro na linha em questao
        (const, msg) = type
        print('ERRO DE SINTAXE [linha %d]: era esperado "%s" mas veio "%s"'
              % (self.current_token.line, msg, self.current_token.lexem))
        #nextToken = self.scanner.getToken()
        #while(not (nextToken.type == self.t.FECHACH or nextToken.type == self.t.PVIRG)):
        #    if nextToken.type == self.t.FIMARQ:
        #        quit()
        #    nextToken = self.scanner.getToken()
        self.current_token = self.scanner.getToken()
        # quit() #mata o programa

```

- Os outros métodos da classe são os métodos que realizam a análise de acordo com as características da linguagem P, por serem vários métodos, fica inviável mostrar todos na documentação, assim sugerimos ao leitor a consulta da gramática da linguagem que está descrita na documentação do trabalho. Os métodos seguem a gramática e inclusive seus nomes são como o não terminais da gramática.

2.3 SymbolsTable.py

Neste arquivo há a classe da estrutura que mantém os id's que são utilizados dentro do programa lido. Essa estrutura no momento só está servindo para fins de armazenamento, será útil de fato somente na implementação do analisador semântico, que verifica o sentido do programa. Essa classe contém alguns métodos:

- Método de inserir um símbolo na tabela:

```

# método para inserir os simbolos na tabela, recebe a key, o tipo, e se há
# parametros ou outros atributos extras
def insert(self, key: str, type, extras: [], line: int):
    # cria um node
    node = self.Node(type, extras, line)
    # e insere ele na tabela utilizando a key
    self.symbols_table[key] = node

```

- Método para remover um símbolo da tabela:

```
# método para remover os simbolos na tabela
def remove(self, key: str):
    self.symbols_table[key] = None
```

- Método para buscar as informações de um símbolo:

```
# metodo para buscar um simbolo na tabela
def get(self, key: str):
    if key in self.symbols_table:
        node = self.symbols_table[key]
        return (node.type, node.extras, node.line)
    return None
```

- Método para limpar a tabela inteira:

```
# metodo para limpar tabela
def free(self):
    self.symbols_table = None
    self.symbols_table = {}
```

- Método para retornar a tabela:

```
#método para buscar a tabela de simbolos
def getSymbolsTable(self):
    return self.symbols_table
```

- Método para mostrar a tabela em console:

```
# método para printar a tabela
def printTable(self):
    print('\t' + 'Chave' + '\t' + 'Tipo' + '\t' + 'Extras' + '\t' + 'Linha')
    for i in self.symbols_table:
        node = self.get(i)
        print('\t' + i + '\t' + str(node[0]) + '\t' + str(node[1]) + '\t' + str(node[2]) + '\n')
```

- Método para exportar a tabela em um arquivo:

```
# método para exportar a tabela para um arquivo externo
def exportToFile(self, fileName):
    file = open(fileName, 'w')
    file.write('\t' + 'Chave' + '\t' + 'Tipo' + '\t' + 'Extras' + '\t' + 'Linha' + '\n')
    for i in self.symbols_table:
        node = self.get(i)
        file.write('\t' + i + '\t' + str(node[0]) + '\t' + str(node[1]) + '\t' + str(node[2]) + '\n')
```

2.4 Outros classes de apoio

As classes de Token (que representa a estrutura do token, que tem seu tipo, o lexema dele e a linha encontrada) e Type (que representa os tipos possíveis de tokens), são classes que são utilizadas pelas outras classes acima, elas não possuem métodos, apenas construtor, por isso não serão descritas com profundidade.

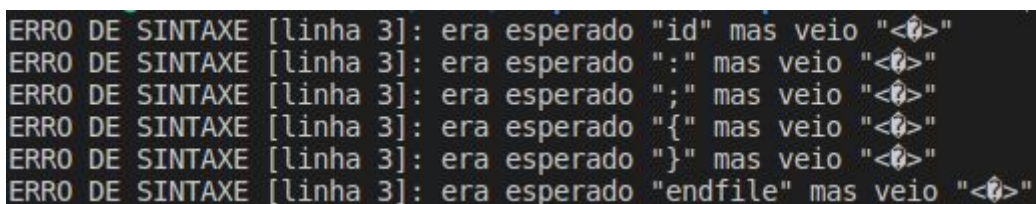
3 - Resultados

Para avaliação dos resultados foram escolhidas duas heurísticas para tratamento de erro, a primeira sendo apenas ao encontrar um erro continuar o trajeto sem utilizar nenhum ponto de sincronia, se a segunda utilizando como ponto de sincronia o token de ponto e vírgula, fecha chaves e fim de arquivo. Assim foi constatado que a execução dos arquivos de teste de 1 a 45, emitiram resposta, alguns emitiram a resposta completamente correta, e outros emitiram com certa precisão, ou seja, emitiram respostas aproximadamente corretas, na maioria dos casos devido a má interpretação de vários erros, assim um dos erros não foi demonstrado ou foi inferido um erro após.

As seções 3.1 e 3.2 mostram como foi os resultados utilizando cada uma das heurísticas apresentada, porém como a quantidade de arquivos era extensa, foi utilizado como base o exemplo de número 15 para apresentação dos resultados ao encontrar um erro durante o processo de compilação.

3.1 Avaliação resultados sem ponto de sincronia

Para esse experimento não foi utilizado nenhum ponto de sincronia ao encontrar um erro, ou seja ao encontrar um erro o compilador continuava seu processo sem se preocupar com o que viria a frente, a imagem abaixo mostra os erros capturados no exemplo de arquivo de teste número 15.



```
ERRO DE SINTAXE [linha 3]: era esperado "id" mas veio "<0>"
ERRO DE SINTAXE [linha 3]: era esperado ":" mas veio "<0>"
ERRO DE SINTAXE [linha 3]: era esperado ";" mas veio "<0>"
ERRO DE SINTAXE [linha 3]: era esperado "{" mas veio "<0>"
ERRO DE SINTAXE [linha 3]: era esperado "}" mas veio "<0>"
ERRO DE SINTAXE [linha 3]: era esperado "endfile" mas veio "<0>"
```

Com base nesse resultado pode se observar que o compilador simplesmente não conseguia seguir em frente, ele ficou completamente parado na linha 3 procurando por símbolos que iriam casar com a regra da gramática, porém nenhum foi encontrado.

3.1 Avaliação resultados com ponto de sincronia

Para esse experimento foi utilizado como ponto de sincronia os tokens ponto e vírgula e fecha chaves, além de obviamente o token de fim de arquivo, fazendo com que o compilador ao encontrar um erro pulasse todos os caracteres seguintes até que encontrasse um ponto de sincronismo, assim a imagem abaixo mostra os erros capturados no exemplo de arquivo de teste número 15.


```
ERRO DE SINTAXE [linha 3]: era esperado "id" mas veio "<>"
ERRO DE SINTAXE [linha 4]: era esperado ":" mas veio "c"
ERRO DE SINTAXE [linha 5]: era esperado ";" mas veio "<$>"
ERRO DE SINTAXE [linha 6]: era esperado "{" mas veio "<>"
ERRO DE SINTAXE [linha 7]: era esperado "}" mas veio "{"
ERRO DE SINTAXE [linha 9]: era esperado "endfile" mas veio "leia"
```

Com base nesse resultado pode se observar que o compilador conseguiu seguir em frente, contudo foi capturado erros que não existiam pelo fato da sequência de tokens recebida não casasse completamente pelo esperado na gramática.

4 - Conclusão

Foi concluído que para o desenvolvimento de um trabalho, o compilador contém o essencial a um trabalho acadêmico, com isso foi possível que para um compilador para uso comercial são necessárias muito validações para impedir que não haja invalidez nos comandos. Para os resultados, como foram utilizadas heurísticas para tratamento de erros capturados, eles foram encontrados da forma esperada para um trabalho acadêmico, então estamos satisfeitos com os resultados encontrados.