Graduação em Ciência da Computação. Disciplinas: “Compiladores I”. Ano 2019.

LÉXICO E SINTÁTICO
RELATÓRIO

Flávia Santos Ribeiro
Luiz Eduardo Pereira

Instituto Federal de Minas Gerais, Formiga, MG.

INTRODUÇÃO

Este trabalho tem como objetivo o projeto e a implementação dos analisadores léxico, sintático, tabela de símbolos e tratamento de erros para a linguagem de programação P.

IMPLEMENTAÇÃO

Foi utilizado como base os códigos disponibilizados em aula. A partir deles foram criados o analisador léxico e o sintático.

- **LÉXICO**

Para o analisador léxico inicialmente foi criado os tipos de tokens seguindo a especificação. Foi criado um token do tipo erro, para quando encontrar tokens não esperados pela linguagem.

Foram criados os seguintes estados que definem o tipo do token:

1: Pega o primeiro caracter do token para classificar qual tipo ele pode ser e o encaminha para o estado correto, também pega o token fim de arquivo, e ignora tabulações.

2: Trata os tokens do tipo identificador ou palavras reservadas. Caso o token tiver mais que 32 caracteres, retorna o token erro.

3: Trata tokens do tipo CADEIA, ou seja, strings. Se encontrar um “ e não encontrar o ”, retorna um token de erro.

4: Trata tokens da classe CTE, ou seja, inteiros e real. No caso do real, se encontra um “.”, é redirecionado para um estado especial, chamado apos_ponto, para que não aconteça casos de real no estilo “5.5.5”.

5: Trata tokens que possuem símbolos, como =, +, >, etc. O simbolo “/” é uma exceção, e não esta contido nesse estado.

6: Trata tokens do tipo comentário ou divisão. Se encontrar o simbolo /* ou //, trata como comentário. Se encontrar apenas “/”, retorna token divisão.

- **SINTÁTICO**

Para o analisador sintático foram descritos as regras das linguagens. Quando a linguagem possui um não terminal, é feito uma função que executa aquele método. Quando a linguagem possui um terminal, o token daquele tipo é consumido.

Se uma regra possui dois ou mais caminhos de derivações possíveis, é feito um IF verificando qual é o primeiro token para saber qual caminho seguir. Por exemplo:

COMANDOS -> IF | WHILE | READ | WRITE | ATRIB
IF -> se abrepar EXPR fechapar C-COMP H
WHILE -> enquanto abrepar EXPR fechapar C-COMP
…

A função comando verifica qual é o token atual. Se for do tipo “SE”, é escolhido o caminho “IF” para a derivação. Nos casos que a regra pode ser vazio, é utilizado um “pass”, para que aquela regra seja ignorada.

Nos casos que temos todas as regras dentro de “IFs”, mas a regra não contem vazio, como exemplo do COMANDOS, foi feito um ELSE para que a função retorne True. A regra que chamou essa função, se receber True, vai levantar o erro que aquela regra esta vazia. Por exemplo:

```
def LISTA_COMANDOS(self):
    if self.COMANDOS():
        self.erro( 'comando' )
    self.G()

def COMANDOS(self):
    if self.atual_igual( tt.SE ):
        self.IF()
    elif self.atual_igual( tt.ENQUANTO ):
        self.WHILE()
    ...
    else:
      return True
```

Isso se deve ao fato de que na regra LISTA-COMANDOS não pode faltar um comando. Caso isso acontecer, um erro do tipo comando é mostrado ao usuário.

A tabela de símbolos foi implementada utilizando um dicionario, em que a chave é o nome da palavra reservada ou identificador, e o valor é uma tupla que contem as informações de constante, tipo e linha para identificadores e para palavra reservada é uma sequencia numérica. No inicio da execução do programa, todas as palavras reservadas são carregadas na tabela de simbolos. Já para os identificadores, toda vez que um token desse tipo é consumido, se ele não existir na tabela de símbolos, ele é adicionado. Para salvar a tabela de símbolos, basta usar -t nome_saida. Mas o -t deve estar depois do nome do arquivo de entrada do sintático.

Execução: python3 sintatico.py entrada.txt -t nome_saida

Houve um problema em ler arquivos que contém ‘ç’. Mesmo convertendo o arquivo para formato unix, não funcionou. Então, foram removidos os ‘ç’ dos testes.

Para o tratamento de erro por modo panico, foi definido um ‘try exception’ em todas as regras. Caso em algum momento houver um erro naquela regra, um erro de “era esperado X, mas veio Y” é lançado junto com um ‘raise’. O raise ativa o exception, que encaminho o programa para a função “sincroniza”. Essa função acessa a tabela de follows de cada regra, e recebe novos tokens até achar um ponto de sincronia. A partir disso, a próxima regra é executada. Note que em muitos casos, após o sincronismo, erros que não existem aparecem, já que apesar de ter sincronizado com um follow, os tokens que aparecem em seguida podem não fazer sentido.

Exemplo do try exception:

```
def A(self):
    try:
        self.PROG()
        self.consome( tt.FIMARQ )
    except:
        quit()
```

CONCLUSÃO

Com este trabalho foi possível concluir na prática o funcionamento da primeira fase de um compilador, os analisadores léxicos e sintático.