Reconhecimento Léxico em Compiladores

Gabriel H. Rudey[1], **Jefferson A. Coppini**[1], **Jonathan T. Rauber**[1], **Nicholas S. Brutti**[1], **Ricardo A. Müller**[1]

[1]Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 - CEP 89802-112 – Chapecó – SC – Brazil

Resumo. *Este trabalho descreve o processo de implementação de um analisador léxico na linguagem Python como trabalho da disciplina de Compiladores da Universidade Federal da Fronteira Sul sob orientação do professor Braulio Adriano de Mello.*

1. Introdução

Segundo [AHO et al. 2007], um compilador é programa que lê um programa escrito em uma linguagem, chamada fonte, e o traduz em um programa equivalente em uma outra linguagem, chamada alvo. Existe uma variedade muito grande de compiladores, mas a forma como estes são organizados, de uma forma geral, segue-se um padrão, dividindo o processo de compilação em duas partes, a análise e a síntese. A análise por sua vez é dividida em três partes, a análise léxica, sintática e semântica.

Este trabalho então, tem como objetivo a implementação de um analisador léxico, escrito na linguagem de programação *Python*. Para melhor compreensão, este trabalho foi dividido em seções, onde primeiramente a seção 2 trata das definições básicas para entendimento do projeto, entre elas a definição do analisador léxico, a seção 3 define características do projeto, como algumas definições básicas do mesmo e um exemplo de código-fonte válido, enquanto a seção 4 apresenta os detalhes de como foi realizada a implementação, dividida em subseções.

2. Referencial Teórico

A função de um analisador léxico, Segundo [PRICE and TOSCANI 2008], é "Fazer a leitura do programa fonte, caractere a caractere, e traduzi-lo para uma sequência de símbolos léxicos, também chamados tokens".

Sendo assim, o analisador léxico deve então fazer a leitura do programa e gerar sobre ele os tokens, que serão utilizados no reconhecimento sintático. [PRICE and TOSCANI 2008] também exemplifica tokens (ou símbolos léxicos) como palavras reservadas, identificadores, constantes e operadores de linguagem, para facilitar a compreensão.

Importante ressaltar que toda informação sobre os tokens é registrada em um estrutura de dados, conhecida como tabela de símbolos. Essa estrutura é extremamente importante para o funcionamento do compilador, pois interage diretamente com todas as etapas do processo de compilação, seja para consulta ou inserção. Uma entrada na tabela de símbolos é da forma: <*nome_token*, *valor_atributo*>. Onde *nome_token* é um símbolo abstrato que representa o identificador, e *valor_atributo* aponta para a entrada correspondente na tabela de símbolos.

A tabela de símbolos está presente na análise semântica, porque armazena informações de identificadores (constantes, variáveis, funções e etc) e o analisador semântico utiliza-se destas informações pois esta é a etapa que verifica operações e dados utilizados, sendo assim, precisa, por exemplo, identificar qual a estrutura de uma variável e se a operação realizada para ela é válida. Sendo assim o analisador semântico necessita das informações dos identificadores, presente na árvore de símbolos, para poder gerar a análise semântica.

Outro ponto relevante à ser definido, são os autômatos não-determinísticos. Para definir um autômato não-determinístico, definiremos primeiramente um autômato determinístico, que, segundo [SIPSER 2007], trata-se de uma máquina onde para um determinado estado e uma determinada entrada, há apenas um estado subsequente. Sendo assim, em um autômato não-determinístico várias escolhas podem existir para um próximo estado em qualquer ponto.

3. Características do Projeto da Linguagem

A linguagem a ser submetida à análise léxica e sintática (esta última análise em um trabalho futuro) foi projetada para suportar estruturas aninhadas de código, como repetição e condição, além de outras operações como declarações de variável e atribuições. É composta por:

- **Constantes**: TRUE, FALSE e MAX;
- **Variáveis**: compostas por n letras minúsculas, com $n > 0$;
- **Numerais**: cadeias compostas por n dígitos de 0 à 9, com $n > 0$;
- **Símbolos especiais**: $+$, $-$, $*$, $/$, $\%$, $(,)$, $\{,\}$, =, ==, !=, $>$, $<$, $<=$, $>=$ e ; (ponto e vírgula);
- **Operadores lógicos**: && (and) e || (or);
- **Tipos**: *integer*, *real* e *boolean*;
- **Palavras reservadas às estruturas da linguagem**: *if*, *else*, *while* e *return*.

O código 1 é um exemplo de código-fonte válido lexicamente e sintaticamente.

```
integer a;
integer b;
a = 10;
b = 5;
if (a <= b) {
  a = b * (a - 999) / a;
} else {
  while (TRUE && a > b) {
    a = a - 1;
  }
}
return;
```

Código-fonte 1: Exemplo de estrutura da linguagem projetada.

4. Implementação e resultados

O processo de analise léxica de uma linguagem é um processo trabalhoso, que requer alguns métodos de desenvolvimento para tal funcionamento. O método de implementação

e técnicas utilizadas durante o processo encontram-se descritos nas subseções abaixo.

4.1. Entrada

A entrada é composta por dois arquivos: o arquivo correspondente ao carregamento dos tokens (entrada.txt) e outro referente ao código fonte do arquivo (fonte.txt).

No arquivo de carregamento encontram-se os tokens da linguagem propriamente ditos e as gramáticas regulares. As gramáticas regulares são as responsáveis por determinar como variáveis e identificadores serão formados.

O arquivo fonte é o responsável pelo programa em si. Nele estão as instruções que serão interpretadas pelo analisador léxico, ou seja, o arquivo possui as instruções do código, separadas por linha (uma instrução por linha).

4.2. Leitura e armazenamento no AFND

Na etapa de armazenamento dos tokens em um autômato não determinístico, a ideia é carregar esses tokens conforme descrito na subseção 4.1, gerando assim as transições necessárias no processo de validação dos tokens.

Nesta etapa o automato é dito não-determinístico pelo fato de possibilitar a existência de mais de uma transição de estados pelo mesmo atributo mas para estados diferentes, como definido na seção 2. Esta situação ocorre apenas no estado inicial, pois é no mesmo que é produzido o primeiro caractere de cada token da linguagem.

4.3. Determinização

Dado um autômato não-determinístico gerado pelo carregamento dos tokens, a missão a partir deste momento é encontrar um modo de eliminar a indeterminização. O processo de determinização surge como solução para o problema.

O processo de determinização consiste em gerar novos estados a partir do estado inicial, estados estes que estejam livres de indeterminismo, ou seja, estados que possuam apenas uma transição pelo mesmo atributo. Ao fim deste processo obtemos um autômato determinístico.

4.4. Minimização

O processo de minimização de um autômato determinístico consiste basicamente em eliminar os estados mortos e inalcançáveis.

Os estados mortos são aqueles que, dada uma derivação, não conseguem alcançar um estados final. A implementação de remoção destes estados do autômato é feita de forma a verificar um vetor de estados alcançáveis a partir de cada estado. Se nesse vetor não for encontrado nenhum estado final, ele é eliminado.

A etapa de eliminar os estados inalcançáveis é a forma de excluir aqueles estados que não são alcançáveis a partir do estado inicial. Na aplicação em questão essa etapa não é necessária, pois esse processo esta implícito durante o processo de determinização, no qual os estados são gerados a partir do estado inicial.

4.5. Análise léxica

Dados os conceitos já abordados, a analise léxica surge como a principal etapa desse desenvolvimento. O objetivo é fazer a análise de um arquivo fonte com várias instruções, verificando se cada cadeia de caracteres pertencente a ele está vinculada a um item que foi carregado durante o processo de inicialização (carregamento de tokens da linguagem).

A aplicação inicialmente interpreta cada linha do arquivo fonte, faz a separação do código fonte cadeia por cadeia e manda cada string separadamente para o reconhecimento léxico. No reconhecimento léxico, a string é percorrida e cada um de seus caracteres é verificado no autômato finito. Se ao chegar no fim da string o reconhecedor tiver alcançado um estado final do autômato, o token é reconhecido.

4.6. Tabela de símbolos

A tabela de símbolos foi implementada como uma lista encadeada de objetos do tipo *token*. Em cada token é verificado se o AFND reconhece o mesmo, dessa forma, caso o retorno seja positivo, é possível afirmar que trata-se de um token válido e portanto deve ser inserido na tabela de símbolos. Caso contrário a operação é abortada, pois o token possui caracteres que não pertencem ao alfabeto da linguagem (erro léxico). A estrutura da classe *token* é a seguinte:

```
class token:
    def __init__(self):
        self.token = ""
        self.cod = -1
        eh_token = False
        self.linha = -1
```

Código-fonte 2: Definição da classe token.

Ao final da análise léxica o algoritmo retorna a fita de saída, que contém todos os tokens reconhecidos e o seu tipo. Caso ocorra algum erro léxico durante o processo, uma mensagem de erro é exibida contendo a linha e o token não reconhecido.

4.7. Validação

Para certificar o funcionamento da solução proposta foram efetuados testes. Os tokens da linguagem foram definidos para suportar a utilização de estruturas condicionais, laços de repetição, declaração de variáveis, operadores lógicos e aritméticos, assim como qualquer linguagem de programação existente. O resultado foi satisfatório, pois o algoritmo foi capaz de cumprir com sua proposta de verificar lexicamente o arquivo de entrada.

5. Conclusão

Este trabalho buscou mostrar a implementação de reconhecedor léxico, desenvolvido sobre a linguagem *Python*, realizando, a partir da leitura sequencial de um arquivo fonte a geração dos tokens, para que em um projeto futuro possam ser desenvolvidas as partes subsequentes de um compilador (como o analisador sintático e semântico).

Referências

AHO, A. V., SETHI, R., and LAM, M. (2007). *Compiladores: Princípios, técnicas e ferramentas*. Longman do Brasil.

PRICE, A. M. A. and TOSCANI, S. S. (2008). *Implementação de Linguagens de Programação: Compiladores*. Bookman Companhia, 1st edition.

SIPSER, M. (2007). *Uma Introdução à Teoria da Computação*. B Thomson, 2nd edition.