Construção de um analisador léxico em Python

Acácia dos Campos da Terra[1], **Gabriel Batista Galli**[1],
João Pedro Winckler Bernardi[1], **Vladimir Belinski**[1]

[1]Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 – 89.802-112 – Chapecó – SC – Brasil

{terra.acacia, g7.galli96, winckler.joao, vlbelinski}@gmail.com

Resumo. *O presente trabalho descreve a implementação de um analisador léxico em Python para reconhecimento dos tokens de uma linguagem hipotética. Inicialmente será apresentada uma contextualização a respeito de reconhecedores léxicos, seguida de uma breve explanação sobre os conceitos, técnicas e teoremas fundamentais para o desenvolvimento do trabalho. Em sequência, serão expostos os detalhes de especificação do analisador léxico, sua implementação e validação. Por fim, as conclusões acerca do trabalho serão exibidas.*

1. Introdução

Segundo [Price and Toscani 2001] o processo de análise léxica consiste na primeira fase de um compilador, quando são identificadas as sequências de caracteres que constituem unidades léxicas. Nessa etapa, o analisador léxico realiza a leitura do programa fonte (visto como uma sequência de palavras de uma linguagem regular), caractere a caractere, verifica se os caracteres lidos pertencem ao alfabeto da linguagem, descarta aqueles não significativos (como comentários e espaços em branco) e realiza a tradução dessa entrada para uma sequência de símbolos léxicos denominados tokens.

Ademais, o analisador léxico geralmente inicia a construção da Tabela de Símbolos (descrita na próxima seção), assim como envia mensagens de erro caso identifique unidades léxicas não aceitas pela linguagem em questão. Por sua vez, a saída do analisador léxico consiste em uma cadeia de tokens utilizada como entrada do próximo módulo, o analisador sintático.

Além disso, cabe destacar que o analisador léxico muitas vezes corresponde a uma subrotina chamada pelo analisador sintático sempre que esse necessita de mais um token. Todavia, existe ao menos uma divisão conceitual entre os analisadores léxico e sintático, pois isso permite a simplificação e modularização do projeto do compilador e possibilita que os tokens possam ser descritos utilizando-se notações simples, além do fato dos reconhecedores construídos a partir da descrição dos tokens serem mais eficientes e compactos do que aqueles construídos a partir das gramáticas livres de contexto (GLCs).

Por fim, a implementação de analisadores léxicos é feita geralmente através de uma tabela de transição, a qual indica a passagem de um estado a outro pela leitura de um determinado caractere. Outra alternativa consiste na construção de um programa que simula o funcionamento do autômato finito que especifica o analisador léxico para a linguagem considerada.

Nesse contexto, o presente trabalho objetiva descrever a construção de um analisador léxico para reconhecimento dos tokens de uma linguagem hipotética, a qual se

encontra definida no arquivo `glc.txt`. Nele, serão identificadas as sequências de caracteres que constituem tokens, construída uma Tabela de Símbolos e uma fita de saída que poderá ser utilizada como entrada de um analisador sintático.

2. Referencial teórico

A seguir serão apresentados alguns conceitos e técnicas fundamentais para a compreensão da implementação de analisadores léxicos. Para outras definições ou exemplificações acerca de compiladores e linguagens formais e autômatos que fogem do escopo do presente trabalho podem ser consultados [Price and Toscani 2001], [Menezes 2000] e [Furtado 2016].

De acordo com [Price and Toscani 2001], **tokens** constituem classes de símbolos tais como palavras reservadas, identificadores, constantes, operadores da linguagem, entre outros, podendo ser representados internamente por três informações:

- classe do token: representa o tipo do token reconhecido (e.g., identificadores, operadores, separadores, etc.); usualmente compiladores os representam por números inteiros;
- valor do token: depende da classe; em função do valor tokens podem ser divididos em:
  - simples: não possuem um valor associado, pois sua classe os descrevem completamente (e.g., palavras reservadas, operadores e delimitadores);
  - com argumento: têm um valor associado (e.g. identificadores e constantes) e correspondem aos elementos da linguagem definidos pelo programador;
- posição do token: indica o local do texto fonte onde ocorreu o token, informação geralmente usada para indicar o local de erros.

A cadeia de tokens produzida como saída do analisador léxico, também denominada **fita de saída**, pode ser representada segundo os itens a seguir:

- identificadores e constantes numéricas são representados por uma terna [classe_token, posição_do_token, índice_tabela_de_símbolos];
- as classes para palavras reservadas constituem-se em abreviações dessas, não sendo necessário passar seus valores para o analisador sintático;
- para delimitadores e operadores a classe é o próprio valor do token.

No que lhe diz respeito, a **Tabela de Símbolos** é uma estrutura de dados montada pelo compilador e que objetiva armazenar informações sobre os nomes (e.g., identificadores de variáveis) definidos no programa fonte. Logo, associa atributos (como tipo) aos nomes definidos pelo programador. Usualmente começa a ser gerada durante a análise léxica, sendo que na primeira vez que um identificador é encontrado esse é então armazenado na tabela, nem sempre sendo possível associar atributos a ele nessa fase. Ademais, conforme [Price and Toscani 2001, p. 27], "toda vez que um identificador é reconhecido no programa fonte, a Tabela de Símbolos é consultada a fim de verificar se o nome já está registrado; caso não esteja, é feita sua inserção na tabela".

Segundo [Price and Toscani 2001, p.17], "uma **linguagem** é um conjunto de palavras formadas por símbolos de um determinado alfabeto", sendo que um **alfabeto**, segundo [Furtado 2016, p. 4], "é um conjunto finito, não vazio, de símbolos (elementos)".

Por sua vez, "uma **sentença** sobre um alfabeto é uma sequência (ou cadeia) finita de símbolos do alfabeto"[Furtado 2016, p. 4].

Tokens de uma linguagem de programação constituem uma linguagem regular, as mais simples segundo a classificação de Chomsky. Uma **linguagem regular** é uma linguagem gerada por uma gramática regular.

Para [Furtado 2016, p. 8], uma **gramática** G é formalmente definida como sendo um quádrupla $G = (Vn, Vt, P, S)$ onde:

- Vn: é um conjunto finito de símbolos denominados não-terminais (símbolos utilizados na descrição da linguagem);
- Vt: é um conjunto finito de símbolos denominados terminais (símbolos da linguagem propriamente ditos, que podem ser usados na formação das sentenças da linguagem);
- P: é um conjunto finito de pares (α , β) denominado produções (ou regras gramaticais ou regras de sintaxe). Uma produção é representada por $\alpha ::= \beta$, onde $\alpha \in V^*VnV^* \wedge \beta \in V^*$, e significa que α é definido por β, ou ainda que α produz β ou equivalentemente que β é produzido (gerado) a partir de α;
- S: é o símbolo inicial da gramática; deve pertencer a Vn. O símbolo inicial de uma gramática é o não-terminal a partir do qual as sentenças de uma linguagem serão geradas.

Ainda segundo [Furtado 2016, p. 10], uma **Gramática Regular (GR)** G é uma gramática definida por $G = (Vn, Vt, P, S)$, onde $P = A \rightarrow aX | A \in Vn, a \in Vt \wedge X \in Vn \cup \varepsilon$.

Autômatos Finitos são reconhecedores de linguagens regulares. Para [Furtado 2016, p. 13] "Entende-se por reconhecedor de uma linguagem "L", um dispositivo que tomando uma sequência w como entrada, respondem "SIM" se $w \in L$ e "NÃO" caso contrário". Autômatos Finitos podem ser Determinísticos (AFD) ou Não Determinísticos (AFND).

Conforme [Furtado 2016, p. 13], formalmente um **AFD** é definido como sendo um sistema formal $M = (K, \Sigma, \delta, qo, F)$, onde:

- K: é um conjunto finito não vazio de **estados**;
- Σ: é um alfabeto, finito, de entrada;
- δ: é a função de mapeamento (ou função de transição) definida em: $Kx\Sigma \rightarrow K$; a interpretação de uma transição $\delta(q, a) = p$, onde $q \wedge p \in K \wedge a \in \Sigma$, é a seguinte: se o "controle de M" está no estado "q" e o próximo símbolo de entrada é "a", então "a" deve ser reconhecido e o controle passar para o estado "p".
- qo: $\in K$, é o estado inicial
- F: $\subseteq K$, é o conjunto de estados finais

Por sua vez, [Furtado 2016, p. 14] aponta que um **AFND** é um sistema formal $M = (K, \Sigma, \delta, qo, F)$, onde:

- K, Σ, qo e F: possuem a mesma definição dos AFD;
- δ: é uma função de mapeamento, definida em $Kx\Sigma = \rho(K)$; sendo que $\rho(K)$ é um subconjunto de K; isto equivale a dizer que $\delta(q, a) = p1, p2, ..., pn$. A interpretação de δ é que M no estado "q", com o símbolo "a" na entrada pode ir tanto para o estado $p1$ como para o estado $p2$, ..., como para o estado pn.

Finalmente, por **determinização** entende-se o processo pelo qual um autômato, caso for um AFND, será transformado em um AFD equivalente e por **minimização** o processo onde são eliminados **estados mortos** (um estado é dito morto quando a partir dele não é possível alcançar um estado final) e **inalcançáveis** (um estado é dito inalcançável quando a partir do estado inicial não consigo alcançá-lo) de um autômato finito.

3. Implementação e resultados

Em relação à implementação do analisador léxico cabe destacar que inicialmente foram definidos os tokens da linguagem e construída a GLC para avaliar e validar o conjunto de tokens para a sintaxe pretendida. A GLC construída pode ser encontrada no arquivo `glc.txt`, sendo que palavras como "if", "else", "for", "while", "let", "plus", "minus", "and" e "or" foram tomadas como reservadas e "$*$", "$/$", "$<=$", "$<$", "$==$", "$! =$", "$>$" e "$>=$", por exemplo, como símbolos. O caractere especial ε é representado por "" (string vazia). Além disso, é utilizada a notação do Formalismo de Backus-Naur (BNF) para fins de representação, sendo que regras precedidas por "$*$" correspondem a partes de GR na GLC e aquelas precedidas por "$+$" a regras alcançadas a partir delas.

Acerca da codificação, tem-se que o arquivo principal do trabalho é `main.py`. Inicialmente, é realizada a leitura da GLC construída através de uma chamada de `readgr()`, definida em `Grammar.py`. Cabe destacar que a leitura se dá caractere por caractere, tendo sido criada uma máquina de estados para verificar se a GLC tomada como entrada se encontra dentro do padrão esperado. Caso não esteja, um erro do tipo `GrammarError` é lançado.

Sequencialmente, é realizada uma chamada à `builtWith()`, presente em `NDFA.py`. Essa função utiliza outras funções definidas no mesmo arquivo, que juntas são responsáveis pela construção de um AFND único para todos os tokens. A tabela do AFND é uma lista. Cada linha dessa tabela é composta por uma estrutura chamada `OrderedDict`, um dicionário ordenado que mapeia cada estado da gramática às suas produções. Cada produção também é um `OrderedDict`, o qual armazena as possíveis transições pelos caracteres do alfabeto da linguagem. Pelo fato do autômato ser não-determinístico, cada mapeamento de um estado por um caractere da linguagem é uma lista de estados. Vale mencionar que os estados da gramática são traduzidos e representados internamente por números, bem como todos os caracteres do alfabeto da linguagem.

Para a construção do AFND em si, itera-se pela gramática recém lida em busca de terminais nas produções da parte livre de contexto. Ao ser encontrado um terminal, é realizado seu mapeamento na tabela, caractere por caractere, sendo criados os estados conforme necessário. Já na parte regular da gramática, é lido o terminal e seu respectivo não-terminal de transição e criado o mapeamento na tabela. Ao final da construção do AFND, é realizado um tratamento especial do ε. Para tal, é iterado sobre o autômato, marcados como finais todos os estados que produzem ε e removida a transição que foi criada na etapa anterior.

Como próximo passo, é realizada a determinização do AFND. Para isso, é chamada outra função de `NDFA.py`, a `to_dfa()`, que funciona como uma busca em profundidade. São inicializados uma pilha de estados com o estado inicial e um conjunto vazio de estados já visitados. Enquanto a pilha não estiver vazia, o estado atualmente presente em seu topo é retirado e determinizado. A determinização é feita da seguinte

maneira: todos os estados das transições do estado atual por um mesmo caractere são guardados em um conjunto. Esse conjunto será o novo estado no autômato determinizado, pelo qual será mapeado o estado atual pelo caractere em questão. Além disso, se qualquer um dos estados desse conjunto for um estado final no AFND, o novo estado do AFD será marcado como final. Cada novo estado é então adicionado ao topo da pilha para ser processado posteriormente.

Em seguida, se há um arquivo de código de entrada como segundo parâmetro, é chamada a função `parse()` da `Parser.py`. Essa função itera pelo arquivo e transforma seu conteúdo em tokens da gramática. A leitura é feita caractere por caractere, transicionando pelos estados definidos pelo AFD. Ao ser encontrado um separador, é verificado se estamos em um estado final ou se há mapeamento do estado atual por este separador. Em caso negativo para as duas situações, tem-se que foi encontrado um token inválido. Caso contrário, é verificado se não existe mapeamento no estado por este separador: se não há, então é adicionado o estado corrente à fita de saída e retornado ao estado inicial; se há e se estivermos num estado inicial ou foi lido um separador na transição anterior, então é realizada a transição existente no estado atual por este separador e marcado que foi lido um separador nesta transição (esta marcação será utilizada aqui novamente para satisfazer a condição recém descrita de caso não estivermos no estado inicial, mas lermos um separador). Se não estivermos sobre um separador, simplesmente é realizada a transição do estado atual por este caractere, caso exista. Senão, se lermos um separador na transição anterior e houver transição por este caractere no estado inicial, é preciso adicionar o estado atual à fita de saída e fazer tal transição (isso serve para tratar tokens compostos por vários caracteres que são separadores). Caso todas as condições anteriores falhem, então é lançado um erro léxico.

Ademais, cabe mencionar como é realizada a construção da Tabela de Símbolos, a qual é montada através de chamadas à função `add()` de `Parser.py`, realizadas pela função `parse()`. Sendo construída dinamicamente, tem-se que em sua primeira posição (0) é montada a fita de saída e nas demais armazenados rótulos de tokens cujo valor seja com argumento. Ao ser reconhecido um token e buscado realizar uma inserção na Tabela de Símbolos algumas verificações são realizadas com base nos parâmetros que a função recebe - estado final encontrado, rótulo e linha de ocorrência:

- caso seja a primeira vez que o rótulo em questão aparece e o estado final encontrado corresponda a um token cujo valor seja simples (e.g., palavras reservadas, operadores e delimitadores) é realizado um `append` na posição (0) da tabela (correspondente à fita) no formato [estado_do_token, linha_de_ocorrência]. De fato, o estado mencionado irá corresponder à classe do token e a linha de ocorrência à sua posição, ambos cuja importância já foi destacada na seção de referencial teórico;
- caso seja a primeira vez que o rótulo em questão aparece e o estado final encontrado corresponda a um token cujo valor seja com argumento (e.g., identificadores e constantes) é realizado um `append` na posição (0) da tabela (correspondente à fita) no formato [estado_do_token, linha_de_ocorrência, índice_tabela_de_símbolos], onde o índice corresponde ao tamanho da tabela em si, que aumenta dinamicamente. Ademais, é adicionado em um conjunto chamado `added`, para o rótulo em questão, o índice do token. Por fim, é adicionado na Tabela de Símbolos no índice em questão o rótulo do token;

- caso não seja a primeira vez que o rótulo em questão aparece, ou seja, caso já exista uma entrada em `added` para o rótulo, é realizado um `append` na posição (0) da tabela no formato [estado_do_token, linha_de_ocorrência, entrada_do_rótulo_em_added]. Note que a entrada do rótulo em `added` corresponde ao índice do token na tabela. Logo, se a variável "x", por exemplo, aparecer diversas vezes em uma entrada, em todas será mapeada para o mesmo índice na Tabela de Símbolos.

Por fim, é importante destacar que para a verificação do funcionamento do analisador léxico implementado foram utilizadas as entradas presentes nos arquivos `1.in`, `2.in` e `3.in`, os quais simulam situações diversas, e verificado se as saídas produzidas por tais entradas correspondiam ao que se esperava, o que de fato ocorre.

4. Conclusões

Ao término desse trabalho conclui-se que a aplicação desenvolvida atende ao objetivo ao qual se propõe, consistindo em um analisador léxico desenvolvido em Python.

No trabalho foi descrita a implementação de um analisador léxico em Python para reconhecimento dos tokens de uma linguagem hipotética, apresentada uma contextualização a respeito de reconhecedores léxicos, uma breve explanação sobre os conceitos, técnicas e teoremas fundamentais para o desenvolvimento do trabalho, expostos os detalhes de especificação do analisador léxico, sua implementação e validação.

Por fim, como perspectivas para a continuidade do trabalho sugere-se o desenvolvimento de uma interface gráfica que, fazendo uso do que é produzido em cada etapa do trabalho, possibilite ao usuário visualizar de uma forma mais agradável tais saídas e selecionar, por exemplo, qual delas deseja acompanhar. Dessa forma, a aplicação se tornaria uma ferramenta de extrema valia no ensino de compiladores, ao passo que permitiria o acompanhamento do processo de análise léxica de uma forma mais facilitada.

Referências

Furtado, O. J. V. (2016). Linguagens formais e compiladores. https://www.ime.usp.br/∼jef/tc_gramaticas.pdf. Acesso em 04/11/2016.

Menezes, P. F. B. (2000). *Linguagens formais e autômatos*. Editora Sagra Luzzatto, 3rd edition.

Price, A. M. A. and Toscani, S. S. (2001). *Implementação de Linguagens de Programação: Compiladores*. Editora Sagra Luzzatto, 2nd edition.