Construção de um analisador sintático 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 sintático em Python para reconhecimento de uma linguagem hipotética cujos tokens já foram definidos no trabalho "Construção de um analisador léxico em Python". Inicialmente será apresentada uma contextualização a respeito de reconhecedores sintáticos, 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 sintático, sua implementação e validação. Por fim, as conclusões acerca do trabalho serão exibidas.*

1. Introdução

Segundo [Price and Toscani 2001] o processo de análise sintática consiste na segunda fase de um tradutor, quando é verificado se a estrutura gramatical do programa está correta (i.e., se essa estrutura foi formada utilizando as regras gramaticais da linguagem).

Sendo as estruturas sintáticas válidas usualmente especificadas através de uma gramática livre de contexto (GLC), o objetivo do analisador sintático é, dada uma GLC G e uma sentença (programa fonte) s, verificar se s pertence à linguagem gerada por G. Assim sendo, o analisador sintático (também denominado *parser*) recebe do analisador léxico a sequência de tokens que formam a sentença s e produz como resultado uma árvore de derivação para s, se a sentença for válida, ou emite uma mensagem de erro caso contrário. A árvore de derivação consiste em uma árvore que exibe a estrutura sintática do texto fonte, resultante da aplicação das regras gramaticais da linguagem.

Conforme apresentado em [Price and Toscani 2001, p. 29], "a árvore de derivação para s pode ser construída explicitamente (representada através de uma estrutura de dados) ou ficar implícita nas chamadas das rotinas que aplicam as regras de produção da gramática durante o reconhecimento". Geralmente a árvore de derivação não é produzida explicitamente, sendo que em vários compiladores a representação interna do programa resultante da análise sintática é somente "uma árvore compactada (árvore de sintaxe) que visa a eliminar redundâncias e elementos supérfluos. Essa estrutura objetiva facilitar a geração do código que é a fase seguinte à análise" [Price and Toscani 2001, p. 9].

Ademais, cabe destacar que os analisadores sintáticos devem ser projetados de modo que possam continuar a análise até o final do programa, inclusive caso venham a encontrar erros de sintaxe no arquivo fonte, os quais devem ser detectados e identificados (posição e tipo).

Existem duas estratégias básicas para a análise sintática:

- Top-Down ou Descendente: a árvore de derivação é construída a partir do símbolo inicial da gramática (raiz da árvore), o que a faz crescer até atingir suas folhas; em cada passo, um lado esquerdo de produção é substituído por um lado direito (expansão);
- Bottom-Up ou Redutiva: a análise é realizada no sentido inverso, ou seja, a partir dos tokens do fonte (folhas da árvore de derivação) a árvore é construída até o símbolo inicial da gramática; em cada passo, um lado direito de produção é substituído por um símbolo não-terminal (redução).

Nesse contexto, o presente trabalho objetiva descrever a construção de um analisador sintático em Python para reconhecimento de uma linguagem hipotética cujos tokens já foram definidos no trabalho "Construção de um analisador léxico em Python". Nele, serão consideradas as estruturas sintáticas definidas no arquivo `glc.gold`, utilizado um sistema auxiliar (*GOLD Parser*) para realizar ações como construção do conjunto de itens válidos, conjunto de transições, FOLLOW e Tabela de Análise LALR, realizado o mapeamento dessa tabela para reconhecimento sintático, o gerenciamento da tabela de símbolos (TS) e integração de procedimentos para tratamento de erros, adição de atributos a símbolos na TS, de valores a esses atributos, entre outros.

2. Referencial teórico

A seguir serão apresentados alguns conceitos e técnicas fundamentais para a compreensão da implementação de analisadores sintáticos. Para outras definições ou exemplificações acerca de compiladores e linguagens formais e autômatos que não foram expostas devido a limitação de tamanho 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. 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.

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].

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.

Uma **Gramática Livre de Contexto (GLC)** é qualquer gramática cujas produções são da forma $A \rightarrow a$, onde $A \in Vn$ e $a \in (Vn \cup Vt)^*$. Segundo [Price and Toscani 2001, p. 30], "a denominação livre de contexto deriva do fato de o não-terminal A poder ser substituído por a em qualquer contexto, isto é, sem depender de qualquer análise dos símbolos que sucedem ou antecedem A na forma sentencial em questão".

Para [Price and Toscani 2001, p. 48], "a função FOLLOW é definida para símbolos não terminais. Sendo A um não-terminal, **FOLLOW(A)** é o conjunto de terminais a que podem aparecer imediatamente à direita de A em alguma forma sentencial. Isto é, o conjunto de terminais a, tal que existe uma derivação da forma $S \Rightarrow *\alpha Aa\beta$ para α e β quaisquer.

Analisadores LR (*Left to right with Rightmost derivation*), conforme [Price and Toscani 2001, p. 66], "são analisadores redutores eficientes que leem a sentença em análise da esquerda para a direita e produzem uma derivação mais à direita ao reverso considerando k símbolos sob o cabeçote de leitura". Existem três tipos de analisadores LR:

- **SLR** (*Simple LR*): de fácil implementação, mas aplicáveis a uma classe restrita de gramáticas;
- **LR Canônicos**: são os mais poderosos, podendo ser aplicados a um grande número de linguagens livres do contexto;
- **LALR** (*Look Ahead LR*): analisadores de nível intermediário e implementação eficiente que funcionam para a maioria das linguagens de programação.

Conforme [Price and Toscani 2001], a estrutura genérica de uma analisador LR consiste em uma fita de entrada, que mostra a sentença $(a_1...a, ...a_n, \$)$ a ser analisada, uma pilha que armazena símbolos da gramática (X) intercalados com estados (E) do analisador e uma Tabela de Análise. O símbolo da base da pilha é E_0, estado inicial do analisador. Por sua vez, o analisador é dirigido pela Tabela de Análise, uma tabela de transição de estados formada por duas partes: a parte AÇÃO contém ações (empilhar, reduzir, aceitar, ou condições de erro) associadas às transições de estados; e a parte TRANSIÇÃO contém transições de estados com relação aos símbolos não-terminais.

O analisador funciona da seguinte forma: seja E_m o estado do topo da pilha e a o token sob o cabeçote de leitura. O analisador consulta a tabela AÇÃO[E_m, a] que pode assumir um dos valores:

- empilha E: causa o empilhamento de "a, E";
- reduz n (onde n é o número da produção $A \rightarrow \beta$): causa o desempilhamento de $2r$ símbolos, onde $r = |\beta|$, e o empilhamento de "AE_γ" onde "E_γ" resulta da consulta à tabela de TRANSIÇÃO[E_{m-r}, A].
- aceita: o analisador reconhece a sentença como válida;
- erro: o analisador para a execução, identificando um erro sintático.

3. Implementação e resultados

Apresentação dos detalhes de especificação/implementação do analisador sintático e estudo de caso para validação.

• Descrição das estruturas sintáticas da linguagem (GLC) • Construção do conjunto de itens válidos e transições e follow • Construção da tabela de parsing SLR ou LALR • Implementação do algoritmo de mapeamento ta tabela para reconhecimento sintático • Gerenciamento da tabela de símbolos (suporte para as próximas etapas) • Integrar procedimentos (ações semânticas – TDS) para os seguintes itens: -tratamento de erros (pelos menos o tipo de erro e linha de ocorrência); -adição de atributos ao símbolos (id de tokens reconhecidos) na TS; -valores nos atributos; -verificação semântica (ao menos uma verificação); -geração de código intermediário (operações básicas); -otimização de código intermediário. Obs.: não é necessário tratar todas as produções, mas uma implementação de cada item para ao menos uma produção.

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.

4. Conclusões

O que foi feito, dificuldades, resultados finais e perspectivas para continuidade do trabalho (por exemplo, sugerir alterações futuras para utilizar a implementação no ensino de compiladores).

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.