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.

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 integrados 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 [Aho et al. 1995], [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

Em relação à implementação do analisador sintático cabe destacar que o mesmo foi desenvolvido fazendo-se uso do analisador léxico também desenvolvido pelos autores em seu trabalho "Construção de um analisador léxico em Python".

Inicialmente, foram realizadas modificações na gramática presente no arquivo `glc.txt` a fim de adequá-la para utilização no programa *GOLD Parser*, o que acabou por resultar na gramática contida no arquivo `glc.gold`. Após isso, tal arquivo foi passado como entrada para o programa *GOLD Parser*, que, por sua vez, realizou ações como a construção do conjunto de itens válidos, conjunto de transições, FOLLOW e por fim a construção de uma tabela de parsing LALR. Do programa mencionado foram tomados então a tabela de parsing LALR, armazenada no arquivo `lalr.htw`, e o conjunto de regras da gramática, armazenado no arquivo `glcrules.htw`.

Em sequência foram construídas estruturas para armazenar as informações contidas nos dois arquivos mencionados, ambas criadas durante a inicialização da classe `Synt()`. Em `buildRules()` de `Synt.py` é lido o arquivo `glcrules.htw` e armazenado seu conteúdo em uma estrutura (nomeada `rules`) que guarda, para uma regra (correponde ao índice da estrutura), seu nome e tamanho. Por sua vez, em `buildLALR()` de `Synt.py` é lido o arquivo `lalr.htw` e armazenado seu conteúdo em uma estrutura (nomeada `ptable`) que guarda, para um dado estado (corresponde ao índice da estrutura), seus mapeamentos para símbolo, ação e estado.

Em seguida, é realizada no arquivo `main.py` uma chamada à função `reckon()` de `Synt.py`. Tal função é responsável pelo reconhecimento sintático. Nela é inicializada uma pilha contendo somente o estado 0 e feito uso da fita de saída e da tabela de símbolos produzidas na etapa de análise léxica. É importante ressaltar que algumas modificações foram realizadas nessa estrutura a fim de comportar a etapa de reconhecimento e a integração de outros procedimentos: passaram a ser armazenados os rótulos de todos os tokens e adicionado "(EOF)" na última posição da fita.

O procedimento a ser realizado para o reconhecimento consiste no seguinte: é tomada a primeira entrada da fita e verificado se seu estado pertence ao conjunto `INTERESTING`, um conjunto que armazena os estados correspondentes a constantes e variáveis. Caso pertença, isso significa que a *label* para aquela posição da fita deve ser percorrida caractere por caractere na tabela de parsing LALR. Caso contrário, toda a *label* deve ser tomada como um só símbolo. Tal tratamento é necessário devido a forma como o *GOLD Parser* monta sua tabela de parsing LALR.

Tendo sido tomado o cuidado acima é chamada para o caractere ou para toda a *label* a função `parse()` de `Synt.py`. Nessa função é tomado o símbolo do topo da pilha, o qual espera-se que seja um número que representa um estado, e verificado se existe mapeamento desse para o caractere ou *label* passada por `reckon()`. Caso não exista mapeamento é informado então a existência de um erro. Caso exista mapeamento é então verificado qual a ação correspondente:

- 'a': corresponde ao estado de aceitação, ou seja, que a entrada está sintaticamente correta. Uma mensagem de aceitação é apresentada e retornado *True*;
- 'g': corresponde à *go to*. Como 'g' é atingido somente se o símbolo a ser considerado for um estado sua ocorrência se dá unicamente em reduções, descrita nos

próximos itens;

- ‘s’: corresponde à *shift*. Quando a ação é ‘s’ são adicionados à pilha o símbolo da fita e o número do estado corresponde ao mapeamento sendo considerado. Além disso, é retornado *True* e então tomado o próximo símbolo (caractere ou toda uma *label*) para análise;
- ‘r’: corresponde à *reduce*, uma redução. Quando a ação é ‘r’ é tomado o número correspondente à redução considerada, acessada a estrutura `rules` no índice igual a esse valor, verificado o tamanho associado à esse índice e desempilhado o dobro de símbolos da pilha. Em seguida é tomado o estado do topo da pilha e verificado seu mapeamento em `ptable` pelo nome da regra anteriormente acessada em `rules`. Então, caso a ação para o mapeamento for ‘g’, são empilhados o nome da regra e o número do estado armazenado para tal mapeamento;
- ‘e’: corresponde a um erro. O tratamento de erros será descrito mais adiante.

É importante enfatizar que caso um mapeamento não seja encontrado isso representa que há um erro sintático na entrada, pois dado o tamanho da gramática nem todos os erros foram tratados. Nesses casos, a natureza de tal erro é então informada ao usuário.

Acerca da integração de procedimentos, foi adicionado à função `add()` de `Parser.py` um trecho de código que realiza a inserção do atributo de escopo e o seu valor às variáveis. Esse atributo é utilizado na análise semântica para verificar se uma variável já existe ao ser utilizada ou se já foi declarada, ou seja, permite a descoberta de utilização de variáveis não declaradas ou redeclaração de variáveis.

A análise semântica mencionada acima é realizada na função `semantic()` de `Synt.py`. Nela, a fita de entrada é percorrida procurando-se por trechos iniciados por tokens `let` até ‘`;`’, bem como por variáveis. Ao ser encontrado um trecho iniciado pelo token que representa `let`, são adicionadas a um dicionário de variáveis conhecidas todas as variáveis desse comando de declaração que não são conhecidas ou que são conhecidas mas cujo escopo não o é. Caso tanto a variável quanto seu escopo já sejam conhecidos, trata-se de um erro de redeclaração de variável. Além disso, se for encontrada uma referência à uma variável que não foi previamente declarada (e não se trate da sua declaração), trata-se então de um erro de variável desconhecida. Cabe destacar que ambos os erros citados são apresentados ao usuário caso venham a ocorrer.

Em relação à recuperação de erros, essa foi realizada seguindo-se as descrições apresentadas nas seções “3.4.3 - Recuperação de Erros na Análise LR” de [Price and Toscani 2001, p. 79] e “Recuperação de Erros na Análise Sintática LR” de [Aho et al. 1995, p. 109]. Conforme é apresentado por [Price and Toscani 2001, p. 79], “na análise LR, os erros são identificados sempre no momento mais cedo, isto é, na leitura de tokens. Nesse tipo de analisadores, a cadeia de símbolos já empilhada está, com certeza, sintaticamente correta. As lacunas na tabela de AÇÃO representam situações de erro e, como tal, devem acionar rotinas de recuperação”.

Assim sendo, para cada um dos índices da tabela de parsing LALR utilizada no trabalho, correspondentes aos estados, é realizado o seguinte:

- nos índices em que houverem reduções essas são propagadas para as lacunas de mapeamento existentes, pois, segundo [Price and Toscani 2001] os erros serão detectados de qualquer forma nos passos subsequentes (na leitura do próximo token);

- nos índices em que existem apenas ações de empilhar, as lacunas de mapeamento existentes são preenchidas com tratamentos apropriados. Cabe destacar que somente foram realizados alguns tratamentos acerca da produção responsável pelas declarações (`let`), devido a grande quantidade de tratamentos que seriam necessários para abranger todos os casos de erro da gramática utilizada.

As ações mencionadas acima encontram-se implementadas nas funções `buildErrors()` e `parse()`, ambas de `Synt.py`. Ao usuário é informado o erro ocorrido, sua linha e também, ao final, o total de erros ocorridos. Dessa forma, a análise sintática realizada não para no primeiro erro encontrado (ao menos para os casos tratados), mas sim busca continuar a análise de todo o código fonte tomado como entrada.

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`, `3.in` e `4.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 sintático desenvolvido em Python.

No trabalho foi descrita a implementação de um analisador sintático em Python para reconhecimento dos tokens de uma linguagem hipotética cujos tokens foram definidos no trabalho "Construção de um analisador léxico em Python", apresentada uma contextualização a respeito de reconhecedores sintáticos, uma breve explanação sobre os conceitos, técnicas e teoremas fundamentais para o desenvolvimento do trabalho, expostos os detalhes de especificação do analisador sintático, 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 sintática, e também léxica, de uma forma mais facilitada.

Referências

Aho, A. V., Sethi, R., and Ullman, J. D. (1995). *Compiladores: Princípios, Técnicas e Ferramentas*. Editora LTC.

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.