Projeto e implementação das análises léxica e sintática de um interpretador SQL

Andrew M. Silva*, Estela M. Vilas Boas†

Universidade Federal da Fronteira Sul (UFFS) – 2019

Resumo

Através do estudo da teoria no componente curricular de Construção de Compiladores e dando continuidade ao trabalho de Linguagens Formais e Autômatos, este artigo descreve a execução do trabalho prático, o qual possui como objetivo melhorar compreensão dos processos e métodos de compilação através de um projeto prático. O projeto prático desenvolvido consiste na implementação das análises léxica e sintáticas de um interpretador SQL, uma Linguagem de Consulta Estruturada comumente utilizada em bancos de dados relacionais. Ao entender o funcionamento de um compilador, é possível concluir sua importância e como sua criação trouxe maior facilidade para o desenvolvimento de projetos e softwares.

Palavras-chaves: Construção de compiladores, linguagens formais, etapas de compilação, análise léxica, análise sintática, análise semântica, código intermediário, otimização.

Introdução

Compiladores "são ferramentas fundamentais no processo de desenvolvimento de software e sua compreensão contribui para um uso melhor"(VANINI, 2019). Um compilador é um elemento básico de toda linguagem de programação e possui o papel de interpretar o código fonte da linguagem em questão para algo que o processador do computador consiga entender e executar. A linguagem de programação, por sua

*andrewsaxx@gmail.com

†estelavilasboas01@gmail.com

vez, irá definir um conjunto de comandos e estruturas mais compreensíveis a um humano, facilitando o desenvolvimento de softwares e permitindo eles possam ser mais complexos e escaláveis.

Logo, se compiladores e linguagens de programação não existissem, o desenvolvimento de softwares como conhecemos hoje seria completamente inviável, pois seria necessário criar softwares apenas com os poucos comandos que um processador entende e executa.

Portanto, é possível perceber o quão importante é a existência de linguagens de programação e de compiladores. Visto isso, este artigo possui como objetivo explicar brevemente o que é e como funciona um compilador e documentar a implementação de interpretador SQL simples.

1 Referencial teórico

Segundo Aho et al. (2007), "tratamos um compilador como uma caixa-preta que mapeia um programa fonte para um programa objeto semanticamente equivalente". Ou seja, um compilador é um processador de linguagem cujo objetivo é traduzir uma linguagem de programação, sendo ela *Java*, *Python* ou *C*, para a linguagem de máquina. Dessa forma, o computador poderá compreender e executar os comandos descritos pelo algoritmo.

Aho et. al (2007) chama a linguagem de máquina de linguagem objeto, ou, programa objeto. Ela é comumente caracterizada como uma linguagem de "baixo nível"por ser composta por um conjunto muito simples de operações que executadas diretamente pelo processador. Esta linguagem é complexa para compreensão e manipulação, além de variar para de acordo com as arquiteturas dos processadores. As linguagens de programação foram criadas para quebrar essa barreira e proporcionar uma codificação mais simples e compreensível e, com elas, vieram seus respectivos compiladores. O objetivo de um compilador é entender a linguagem de alto nível e traduzí-la ou interpretá-la para linguagem de máquina.

Conforme Aho et. al (2007), ao analisar o funcionamento de um compilador, é possível observar que seu processo é "desenvolvido como uma sequência de fases", cada uma gerando resultados que as seguintes utilizarão. Essa sequência de fases — análises e traduções — foi inserida na proposta de realização do trabalho prático.

A análise efetuada por um compilador "subdivide o programa fonte em partes constituintes e impõe uma estrutura gramatical sobre elas. Depois usa essa estrutura para criar uma representação intermediária do programa fonte"(AHO, 2007, p.3). Caso a análise encontre no programa fonte alguma inconsistência, um código mal estruturado sintaticamente ou semanticamente, o compilador deve fornecer, com o resultado da análise, informações explicativas ao utilizador da linguagem. Dessa forma, o usuário poderá tomar as medidas necessárias para a correção.

Além disso, "a parte da análise também coleta informações sobre o programa fonte e as armazena em uma estrutura de dados chamada tabela de símbolos, que é passada adiante [...] para a parte de síntese"(AHO, 2007, p.3). Portanto, faz-se necessário entender o que constitui cada etapa de compilação, os quais estão descritas

logo abaixo:

- Análise Léxica;
- Análise Sintática;
- Análise Semântica;
- Geração de código intermediário;
- Otimização de código intermediário.

A análise léxica, ou *scanning*, realiza a leitura do código fonte, caractere por caractere, para identificar todos os identificadores, operadores e constantes e verificar eles obedecem às regras gramaticais da linguagem. Esta tarefa é realizada fazendo o uso de uma máquina de estados finitos, ou *Finite State Machine* (FSM), contendo todo o alfabeto da linguagem e as regras gramaticais que a definem. Dessa forma, se alguma sequência de caracteres infringir alguma dessas regras ou possuir um ou mais caracteres que não pertencem ao alfabeto da linguagem, ocorrerá um erro léxico. Caso isso não ocorra, cada identificador reconhecido, também chamado de *token*, é armazenado numa tabela de símbolos e numa fita de saída para ser usado pelas etapas seguintes.

Diferentemente, a análise sintática, ou *parsing*, possui o papel de identificar se a ordem dos *tokens* reconhecidos é válida. Para tal, é necessário possuir uma GLC (Gramática Livre de Contexto) com as regras gramaticais que definem as ordens dos *tokens*. É nela que será definido como um comando *if* é estruturado, por exemplo. Esta GLC é utilizada para a geração de uma tabela de *parsing* que é utilizada para verificar se a sequência dos *tokens* de uma a fita de saída está correta ou não. A tabela de *parsing* utilizada neste trabalho é a SLR (Simple Left-Right), o qual faz a análise sintática a partir de reduções dos símbolos mais à esquerda.

É possível perceber que as duas primeiras etapas prezam pela corretude dos *tokens* sem levar em conta seus possíveis significados. Este fato pode dar a impressão de ambas possuem funções relativamente semelhantes, porém "a separação das duas em dois níveis de descrição e tratamento simplifica não só a descrição da linguagem como também a implementação do compilador"(VANINI, 2019). A mais clara diferença entre as duas etapas é que a análise sintática apenas preza pela ordem dos *tokens* já reconhecidos, enquanto a análise léxica apenas verifica se os *tokens* estão escritos corretamente.

É na análise semântica em que os possíveis significados dos *tokens* serão validados. Se as variáveis da linguagem possuírem tipagem, é nesta etapa que elas serão analisada, podendo gerar error de tipagem ao, por exemplo, tentar atribuir um número real à uma variável do tipo caractere. Para a realização desta etapa, cada sequência sintática válida precisa possuir suas respectivas ações semânticas. Por exemplo, se a linguagem de programação em questão for interpretada para outra linguagem, as ações semânticas muito provavelmente serão executadas nesta outra linguagem.

Quando a linguagem de programação não é um interpretador e gera um código objeto, isto é, em linguagem de máquina, utiliza-se a geração de código de máquina. Esta etapa utiliza-se dos mesmos métodos da análise semântica, mas com a diferença de que, ao invés de executar ações semânticas, ela gerará um código objeto, normalmente utilizando arquitetura de 3 endereços por ser independente de máquina.

Todavia, a sequência de operações presente no código intermediário gerado nem sempre está otimizado e, portanto,

> "alguns compiladores possuem uma fase de otimização[...]. A finalidade dessa fase de otimização é realizar transformações na representação intermediária, de modo que o back-end possa produzir um programa objeto melhor do que teria produzido a partir de uma representação intermediária não otimizada."(AHO, 2007, p.3)

2 Projeto prático

O projeto prático consiste no desenvolvimento de um interpretador simples para SQL (Structured Query Language) num um banco de dados relacional. Este interpretador foi desenvolvido em *python3* e é capaz de interpretar os comandos mais comuns, como criação e remoção de tabelas, inserção, alteração e remoção de dados e consultas.

Para tal, foram desenvolvidas as duas primeiras etapas de compilação, os quais já foram explicadas anteriormente: a análise léxica e a análise sintática. O código fonte do projeto pode ser encontrado no GitHub em *andrewmsilva/SQLInterpreter*. Nas subseções abaixo está descrito como cada etapa foi implementada.

2.1 Análise léxica

A implementação da análise léxica consiste na definição dos *tokens*, regras gramáticas e separadores da linguagem, na geração da FSM determinística e no reconhecimento léxico considerando os *tokens* válidos e os caracteres separadores.

Os *tokens*, as regras gramaticais e os separadores da linguagem foram definidos em três arquivos: *set/tokens.txt*, *set/grammatics.txt* e *set/separators.txt*. Neles, cada linha contém um *token*, uma regra gramatical ou um separador.

Feito isso, criou-se a classe FSM no arquivo *src/FSM.py* que, quando instanciada, lê e mapeia as gramáticas e *tokens* definidos em seus respectivos arquivos, remove as épsilon transições, determiniza a FSM, remove estados inalcançáveis e/ou mortos e mapeia os estados de erro. Por padrão, a FSM gerada é salva em arquivo para ser obtida facilmente nas próximas vezes que o algoritmo for executado, o qual carregará o arquivo automaticamente. Esta classe também fornece métodos úteis, os quais estão descritos a seguir:

- *show*: mostra a FSM no *console*;
- *getInitialState*: retorna o nome do estado inicial;

- *getErrorState*: retorna o nome do estado de erro;
- *makeTransition*: retorna o estado resultado da transição a partir de um outro estado e um caractere, ambos recebidos por parâmetro;
- *isFinal*: verifica se um estado recebi por parâmetro é ou não um estado final.

Após isso, criou-se a classe *Lexer* no arquivo *src/Lexer.py*, o qual estende a classe FSM, carrega os separadores definidos em arquivo e fornece o método *analyze* para reconhecer os *tokens* de uma sequência de caracteres recebidos por parâmetro. Este método gerará a tabela de símbolos como um dicionário de dados e a fita de saída como uma sequência de caracteres, mas, caso haja alguma violação léxica, um erro é mostrado no *console*.

2.2 Análise sintática

A primeira etapa na implementação da análise sintática é a definição da GLC e da SLR. A GLC foi definida para ser capaz de interpretar os comandos SQL mais comuns, como criação e remoção de tabelas, inserção, alteração e remoção de dados e consultas. Feito isso, foi gerada tabela SLR utilizando o gerador presente no site *http://jsmachines.sourceforge.net/machines/slr.html*. Então a GLC e a SLR foram salvas nos arquivos *set/GLC.csv* e *set/SLR.csv*, respectivamente.

Feito isso, criada a classe *Parser* no arquivo *src/Parser.py* que, quando instanciada, carrega a GLC e a SLR definidas em arquivo e fornece o método *parse*, o qual erá realizar a redução dos *tokens* ma fita de saída recebida por parâmetro e verificar se sua ordem é válida de acordo com as regras definidas na GLC.

Por padrão, apenas é mostrado algo no *console* quando um erro sintático ocorre, mas é possível instanciar o *Parser* em modo *degug*. Neste caso, sempre que uma fita de saída foi analisada, será mostrada no *console* a sequência de passos do seu reconhecimento.

2.3 Interpretador

O interpretador SQL está definido no arquivo *SQLInterpreter.py* e consiste em instanciar as classes *Lexer* e *Parser* e fornecer o método *interpret*, o qual realiza as duas etapas de interpretação em uma *query* SQL recebida por parâmetro.

Para testar o interpretador, foi criado o *test.py* que importa o *SQLInterpreter* e interpreta comandos recebidos pelo teclado infinitas iterações. É possível visualizar os erros léxicos e sintáticos no *console* quando existirem.

3 Considerações finais

Os resultados obtidos se mostraram muito promissores e, se mais etapas de compilação fossem implementadas, seria totalmente possível tornar o interpretador funcional. Visto o quão importante é entender o que é e como funciona um compilador,

é evidente como a implementação do projeto prático contribuiu para a compreensão do funcionamento de compiladores e interpretadores. De fato, foram implementadas apenas duas etapas de compilação, mas todas foram implementadas visando facilitar o trabalho das etapas seguintes.

Referências

AHO, A. V. et. al. *Compiladores: Princípios, técnicas e ferramentas*. 2. ed. [S.l.]: Pearson Universidades, 2007. 2, 4

VANINI, F. A. *Construção de Compiladores*. [S.l.], 2019. Disponível em: <http://www.ic.unicamp.br/~vanini/mc910/Parte1.pdf>. Acesso em: 01 dez. 2019. 1, 3