Relatório do Trabalho Final da Disciplina de Compiladores

Diogo Maceil da Cunha[1]

[1]Colegiado de Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 – 89815-899 – Chapecó – SC – Brazil

diogomaciel.cunha@gmail.com

Abstract. *Compilers are programs responsible for reading a user's source code and mapping it to a corresponding machine code. To do this, they use concepts derived from the study of Formal Languages and Automata, seeking to generate structures that facilitate this process, such as Deterministic Finite Automata and Pushdown Automata. This work aims to describe an application that implements the lexical and syntactic analysis stages of compilation, to validate whether a given source file is valid to be interpreted or transcribed to the following stages. In general, the application was able to perform the analyzes but improvements can be made seeking to increase the number of tokens and logical structures that it can recognize.*

Resumo. *Compiladores são programas responsáveis por ler um código fonte do usuário e mapeá-lo para um código de máquina correspondente. Para isso, utilizam conceitos derivados do estudo de Linguagens Formais e Autômatos, buscando gerar estruturas que facilitem esse processo, como os Autômatos Finitos Determinísticos e os Autômatos de Pilha. Este trabalho tem como objetivo descrever uma aplicação que implementa as etapas de análise léxica e sintática de compilação, para validar se um determinado arquivo fonte é válido para ser interpretado ou transcrito para as etapas seguintes. No geral, a aplicação conseguiu realizar as análises mas melhorias podem ser feitas buscando aumentar a quantidade de tokens e estruturas lógicas que ela pode reconhecer.*

1. Introdução

Com o avanço dos computadores, o aumento na quantidade de memória e processamento de cpu possibilitaram a implementação de formas de simplificar o desenvolvimento de programas. Antes, os programas precisam ser escritos com a linguagem de montagem específica da arquitetura da CPU, ou seja, programas que funcionavam em processadores ARMs não funcionavam em X86 e assim por diante. Para resolver esse problema, os compiladores foram concebidos como intermediários entre uma linguagem formal e a linguagem de montagem desejada, assim reduzindo a complexidade da manutenção de código e criando processos de otimização automatizados.

Os compiladores, recebem um arquivo de carga do usuário, e o decodificam em uma linguagem de máquina através de múltiplas etapas de compilação sequenciais, em um processo de linha de montagem que compartilha uma estrutura chamada tabela de símbolos. A tabela de símbolos, armazena todos os dados extraídos da entrada e seus respectivos mapeamentos em tokens e estruturas. É comum que ela também seja utilizada para retornar os erros ao usuário. Dito isso, esse trabalho tem como objetivo descrever

o funcionamento de uma ferramenta que simula duas etapas de compilação, a análise léxica e sintática, utilizando os formalismos de linguagens formais para reconhecimento de *tokens* e estruturas. Nele, é utilizado um autômato finito determinístico (AFD) para o reconhecimento dos tokens da linguagem, e um autômato de pilha *left right* (LR) como tabela de derivação para reconhecimento de estruturas como estruturas de encadeamento lógico e de operações aritméticas.

2. Referencial Teórico

2.1. Gramáticas

Uma gramática é um conjunto formado por um alfabeto e uma função de transição que define um conjunto de cadeias válidas, reconhecidas. No geral, existem três tipos de gramáticas agrupadas pelo seu poder de reconhecimento: regulares, livres de contexto, sensíveis ao contexto. As mais simples são as regulares, que são capazes de gerar um conjunto de cadeias com qualquer quantidade de símbolos, mas apenas gerando um símbolo de cada vez. São muito úteis para reconhecimentos de cadeias e geração de regras de validação de palavras mas não são capazes de reconhecer a ordem dos *tokens* que geram como saída. Por exemplo, caso exista uma gramática regular que reconheça os *tokens*, ”1 + 1 = 2”, para ela essa cadeia é igual a esta ”1 1 2 = +”pois os mesmos *tokens* foram gerados. [Scheffel 2011]

Já uma gramática livre de contexto, é capaz também de reconhecer *tokens* e estabelecer uma ordem em que eles devem estar. Isso acontece por ela conseguir gerar mais símbolos por transição. Assim, usando a mesma cadeia como exemplo, ela não é capaz de criar a cadeia ”1 1 2 = +”mas consegue gerar a cadeia ”true + false = 1”com uma operação com tipagem inválida. E por fim, as linguagens sensíveis ao contexto são as mais poderosas, por fazer tudo que as outras fazem e gerar cadeias com regras bem definidas, como operações aritméticas que utilizem somente números. [Scheffel 2011]

2.2. Autômatos

Já os autômatos, são as estruturas capazes de reconhecer as cadeias geradas pelas gramáticas, são respectivamente: autômato finito, autômato de pilha e máquinas de Turing. Nessa seção, irei me ater aos dois primeiros por serem os que foram diretamente aplicados no trabalho. Um autômato finito é uma estrutura que reconhece as cadeias geradas por uma gramática regular, faz isso através de um mapeamento de transições onde só enxerga o símbolo atual e o utiliza para achar o próximo. Existem apenas dois tipos de autômatos finitos, os determinísticos e os não determinísticos. O primeiro tipo, não possui ambiguidade, por exemplo, não há transição saindo pelo símbolo ’a’ vai para dois estados diferentes, e o segundo pode possuir essa ambiguidade. [Scheffel 2011]

Já os autômatos de pilha, possuem também a classificação de determinísticos e não determinísticos, mas também são divididos pela forma como são montados. Em suma, a diferença entre essa e a categoria anterior é que eles possuem uma memória restrita que armazena o último estado antes do atual e podem executar a ação de empilhar e remover ele, como em uma pilha. A categoria LR, utiliza as produções, o conjunto *first* e *follow* e as derivações possíveis para montar a tabela, ela mapeia três ações possíveis: *shift*, *reduce* e *goto*. A primeira, empilha dois valores na tabela, o token da entrada e a transição que é realizada a partir dele, assim fazendo um empilhamento da estrutura que foi fornecida

como entrada. O segundo, retira valores da pilha, e os substitui pela transição que foi montada pela ação anterior. E o *goto*, troca uma transição por outra, normalmente visando continuar o empilhamento de um novo ponto. Por fim, vale ressaltar que todos os não determinísticos apenas aceitam gramáticas livre de contexto determinísticas, fatoradas, e, os LR, também precisam de gramáticas sem recursão a esquerda. [Louden 2004]

2.3. Etapas de Compilação

As etapas de compilação são os processos que o compilador utiliza para gerar o código de máquina, elas são: análise léxica, análise sintática, análise semântica, geração de código intermediário e otimização. A análise léxica é o primeiro tratamento que o código fonte recebe, nela o código escrito pelo programador é transformado em uma fita de *tokens*, que será consumida pela etapa subsqueste, e em uma tabela de simbolos contendo, o valor original, o token equivalente e a linha onde ele foi lido. Caso haja um simbolo desconhecido no código fonte, o compilador irá reconhecê-lo como erro léxico, interrompirá o processo de compilação e retorná o erro ao usuário.

A análise sintática é onde são reconhecidas as estruturas (condicionais, aritméticas etc) que compoem o código. Para isso, a fita gerada pela etapa anteiror é consumida e se utiliza o autômato de pilha como tabela de derivação para o reconhecimento de cada estrutura. Ao fim, é retornada a tabela de simbolos ou um error, caso seja encontrada uma ordem não permitida nos tokens. A análise sintática não realiza o tratamento semântico ou gerá código intermediário mas pode realizar ações semânticas que geram atributos adicionais na tabela de simbolos que serão utilizadas pelas etapas posteriores.

Na análise semântica, é feito a checagem de tipos dos valores, para impedir que um número se comporte como uma cadeia de caracteres etc. Na geração de código intermediário, é criado um código de três endereços que será consumido no processo de otimização. E na otimização, se busca reduzir o tamanho do código final e melhoram seu desempenho geral, com menos utilização de recursos e mais velocidade na execução. Esse último processo, é um problema NP, por isso são utilizados processos heurísticos para reduzir seu tempo de execução. [Louden 2004]

3. Descrição

3.1. Análise Léxica

Para a análise léxica, foi utilizado um autômato finito determinístico para o reconhecimento dos tokens. O programa só reconhece dois tipos de *tokens*, palavras reservadas e variáveis. Cada palavra reservada é reconhecida por um valor número que é o número do estado em que o autômato a reconhece, que pode variar de 0 a n. Já as variáveis, possuem o marcador ”var”e são geradas por uma gramática regular que gera cadeias formadas por a,b e c. Para gerar a fita, o programa lê o arquivo fonte e o transforma em uma lista de palavras, removendo todos os espaços em branco e tabulações. Depois, percorre cada palavra da lista para encontrar o *token* equivalente. No fim, é retornando a fita e a tabela de símbolos ou erro que identifica em que linha foi encontrado o *token* incorreto e qual a cadeia errada. Para reconhecer um *token*, cada caractere da palavra é passado para o AFD de forma a percorrê-lo até encontrar um estado final ou a palavra acabar.

3.2. Análise Sintática

Para a análise sintática, foi utilizado um programa chamado *Gold Parser* para gerar a tabela de derivação. O resultado foi mapeado em três estruturas da tabela, o mapamento de símbolo para identificador de símbolos, as produções e a tabela de derivação LR. Após a leitura, foi utilizada uma classe com esses três atributos como tabela. Não foram implementadas ações semânticas na aplicação, mas a saída da análise sintática foi a tabela de símbolos ou um erro sintático. No retorno do erro, é apresentado ao usuário a estrutura que causou o erro e a linha onde ele ocorreu.

4. Conclusão

A aplicação funcionou como o esperado e a estrutura final do código foi satisfatória mas houve problemas na identificação dos *tokens* que precisam ser resolvidos. A análise léxica foi o processo que mais apresentou problemas por causa da geração dos estados do AFD e seria interessante mudar a forma como as regras são carregadas para permitir a inserção de mais gramáticas para novos tipos como números, *strings* e etc. Por outro lado, o que apresentou maior dificuldade na análise sintática foram as carências e o mapeamento de estados da análise léxica, mas os processos de análises em si não foram problemáticos e funcionaram após poucas tentativas. Como trabalhos futuros, ficam a resolução de *bugs* na análise léxica, a criação de um nova estrutura de carga de *tokens* e a adição das ações semânticas na análise sintática.

Referências

Louden, K. C. (2004). *Compiladores-Princípios e Práticas*. Cengage Learning Editores.

Scheffel, R. M. (2011). *Apostila Linguagens Formais e Autômatos*. Departamento de Ciencias Tecnologicas, UNISUL, 1 edition.