Construção de uma aplicação para geração, determinização e minimização de Autômatos Finitos em linguagem C++

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

Abstract. *This paper describes the implementation of an application in C++ programming language for generation, determinization and minimization (without the use of equivalence classes) of Finite Automata constructed from tokens and Regular Grammars (RGs). Consisting in a possible version of a lexical analyzer, the particularities and operation of the application, such as its planning and code will be commented and analyzed in the work. At the end it will be checked the operation of the application by conducting tests.*

Resumo. *O presente trabalho descreve a implementação de uma aplicação em linguagem de programação C++ para geração, determinização e minimização (sem utilização de Classes de Equivalência) de Autômatos Finitos construídos a partir de tokens e Gramáticas Regulares (GRs). Consistindo em uma possível versão de analisador léxico, as particularidades e funcionamento da aplicação, tal como seu planejamento e código serão comentados e analisados no trabalho. No final será verificado o funcionamento da aplicação através da realização de testes.*

1. Introdução

O presente trabalho objetiva descrever a construção de uma aplicação em linguagem de programação C++ para geração de um Autômato Finito Determinístico (AFD) a partir de tokens/cadeias e Gramáticas Regulares (GRs), assim como a determinização e minimização do Autômato Finito (AF) sem a utilização de Classes de Equivalência.

Na Seção 2 será descrito um possível processo de construção de analisadores léxicos. Por sua vez, na Seção 3 será realizada uma descrição detalhada da aplicação. Em sequência, a implementação e planejmento da aplicação serão demonstrados na Seção 4 e os testes que verificam seu funcionamento na Seção 5. Por fim, na Seção 6 poderão ser encontradas as conclusões acerca do trabalho.

2. Descrição de um possível processo de construção de analisadores léxicos

Um possível processo de construção de analisadores léxicos objetiva, a partir de tokens, a geração de um Autômato Finito Determinístico (AFD) mínimo e sem Classes de Equivalência que reconhece (ou não) tais tokens como símbolos de uma linguagem. Por AFD mínimo têm-se o autômato com o menor número de estados para a linguagem, como pode ser visto em [Menezes 2000, p. 72].

Para isso, inicialmente um conjunto de tokens, que podem ser cadeias ou GRs, é considerado. Para cada token é realizada então a construção de um AF seguida da composição desses Autômatos Finitos gerados em um único autômato.

Em sequência à composição dos AFs gerados em um único AF é realizada a determinização, processo pelo qual o autômato, se for um Autômato Finito Não Determinístico (AFND), será transformado em um AFD equivalente. Em seguida, ocorre a etapa de minimização, 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).

Todavia, não é feito uso de Classes de Equivalência, pois é necessário que cada estado final reconheça um conjunto de sentenças específico, sendo que a utilização de Classes de Equivalência poderia vir a unir estados que reconhecem situações diferentes, podendo a versão minimizada até mesmo reconhecer sentenças que antes não o eram.

Por fim, a respeito da construção de analisadores léxicos temos o algoritmo de análise léxica, onde existe uma condição que é utilizada na verificação de cada símbolo lido, checando-se se tal símbolo é um separador (pode ser um espaço, quebra de linha...) ou não e, caso for separador, se também é token. Nessa verificação têm-se que ao encontrar um símbolo separador é tomado o estado em que se parou e inserido esse na fita de saída. Por sua vez, a fita de saída será utilizada como entrada para um possível analisador sintático. Cabe ressaltar que se o separador for também um token é necessário prosseguir na análise para saber se de fato naquele momento de leitura o símbolo era parte de um token ou um separador.

3. Descrição da Aplicação

A aplicação desenvolvida faz a carga de tokens e Gramáticas Regulares na notação do Formalismo de Backus-Naur (BNF, do inglês *Backus-Naur Form* ou *Backus Normal Form*) a partir de um arquivo fonte (texto). Para cada token e gramática é gerado um conjunto de transições rotuladas em um único AF durante a carga. Para tal, é compartilhado apenas o estado inicial, sendo gerados estados exclusivos para as transações dos demais símbolos dos tokens e/ou estados das Gramáticas Regulares. O AF gerado será não determinístico quando acontecer um ou mais casos em que dois tokens ou sentenças definidas por gramáticas regulares iniciam pelo mesmo símbolo.

Após a construção do AFND é aplicado o teorema de determinização para obter o AFD equivalente. Por sua vez, o AFD resultante é submetido ao processo de minimização sem aplicar Classes de Equivalências, devido ao motivo destacado na seção anterior. Em sequência, as transições não mapeadas são preenchidas por estado de erro, sendo que a coluna final "x" representa todos os símbolos não pertencentes ao conjunto de símbolos de todos os alfabetos das linguagens representadas por tokens e gramáticas.

Por fim, no AFD final os estados são (REPRESENTADOS POR NÚMEROS?) e os símbolos por seu correspondente numérico de acordo com a tabela ASCII (??), sendo o AFD final é salvo em um arquivo de saída no formato `.csv`.

4. Planejamento e Implementação

A implementação da aplicação pode ser encontrada em três arquivos: `ndfa.cpp`, `automata.h` e `automata.cpp`.

No arquivo `ndfa.cpp` se encontra a função `main`, onde é realizada a leitura do arquivo de entrada e verificado se o mesmo está no padrão considerado pela aplicação. Cabe ressaltar que Gramáticas Regulares devem ser precedidas por uma linha contendo o número 0. Por sua vez, tokens devem ser precedidos por uma linha contendo a quantidade de tokens que seguirão. Cabe destacar que cada token, por exemplo, deve ser seguido por uma quebra de linha, enquanto a separação de blocos de gramáticas e tokens deve ser feita por meio de duas quebras de linha.

Na função main ainda é realizada a impressão de mensagens no prompt a fim de permitir ao usuário o acompanhamento da aplicação e realizadas chamadas às funções responsáveis pelas ações executadas pela aplicação.

No arquivo `automata.h` podem ser encontradas as definições das constantes utilizadas no trabalho, das estruturas de dados e os protótipos das funções implementadas em `automata.cpp`.

Em `automata.cpp` têm-se uma estrutura, nomeada `enumstate` e 11 funções, as quais serão explicadas uma a uma a seguir.

Inicialmente, em relação à `struct enumstate` cabe destacar que sua funcionalidade consiste em....

–DESTACAR PARÂMETROS, FUNCIONALIDADE, ESTRUTURAS UTILIZADAS, RETORNO/SAÍDA DE CADA UMA DS FUNÇÕES ABAIXO:

A função `debugprint`...

Por sua vez, a função `debugd`...

Em relação à `readgrammar` têm-se que...

A função `readtokens` recebe com entrada um inteiro correspondente a...

Para.. foi implementada a função `debugf`, que...

No que lhe diz respeito, a função `makedet`...

Por sua vez, a função `minimize` possui duas assinaturas. Em uma delas recebe como entrada uma `string` correspondente a... Na outra não apresenta parâmetros, servindo para...

A respeito da função `remove` têm-se que...

A função `fill` foi implementada a fim de...

Por fim, a função `csv`...

5. Testes

Para a realização dos testes foram criados dois arquivos. O primeiro, `test.in`, apresenta como entrada de teste uma única Gramática Regular. Por sua vez, o segundo arquivo de testes, nomeado `exemple.in`, testa tanto GRs quanto tokens. Para isso, contém uma Gramática Regular seguida por seis tokens e por mais uma GR.

A partir da realização dos testes pôde ser verificado que os resultados obtidos condizem com as saídas esperadas, essas que podem ser verificadas no arquivo `.csv` gerado. Isso demonstra que a aplicação construída atende aos objetivos a qual se propõe.

6. Conclusão

Texto...

Referências

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