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). Possibly being part of a lexical analyzer, are going to be commented and analyzed in the paper the particularities and operation of the application, such as its planning and code. 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). Possivelmente constituindo parte de um analisador léxico, serão comentadas e analisadas no trabalho as particularidades e funcionamento da aplicação, tal como seu planejamento e código. 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 da aplicação e seu planejamento. Em sequência, a implementação da aplicação será demonstrada 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 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 e Planejamento 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 letras, sendo que a nomeação dos estados se dá através da enumeração usual do alfabeto. Em relação aos símbolos não foram realizadas modificações. Como último passo, o AFD resultante de todo o processo executado é salvo em um arquivo de saída no formato `.csv`.

4. 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. Além disso, é importante 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 no processo proposto.

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`. No que lhe concerne, o arquivo `automata.cpp` contém 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 atribuir novos nomes aos estados tomados do arquivo de entrada. A nomeação dos estados é feita de acordo com a enumeração usual do alfabeto (excetuando-se 'S' e 'X', que são reservados aos estados inicial e estado de erro, respectivamente).

A função `debugprint` realiza a impressão do autômato recebido como entrada em notação BNF. Por sua vez, a função `debugd` imprime o autômato determinizado ou minimizado, também na BNF.

Em relação à `readgrammar` tem-se que sua funcionalidade consiste na leitura de uma Gramática Regular na notação BNF e construção de seu respectivo autômato finito, que ainda pode ser não determinístico. Para isso, como os estados são renomeados segundo a enumeração usual do alfabeto foi criado um `map` denominado `states` de forma a mapear o nome original do estado com o novo nome. Como a gramática está no formato BNF a identificação dos estados é feita a partir da presença do seu nome entre os símbolos de '$<$' e '$>$'.

A função `readtokens` recebe com entrada um inteiro correspondente a quantidade de tokens que serão lidos. Sua tarefa consiste na leitura de tokens e construção de seu respectivo autômato finito, que tal como em `readgrammar` ainda pode ser não determinístico. Para fins de acompanhamento e verificação também foi implementada a função `debugf`, que imprime os estados finais do autômato.

No que lhe diz respeito, a função `makedet` é responsável pela determinização do autômato gerado por `readgrammar`. Nela foi utilizado um vetor de `string`s nomeado `states`, como uma fila, para indicar quais estados serão determinizados. É iniciado, então, pelo estado inicial S e seguido realizando a determinização de todos os estados alcançáveis a partir do estado atual, colocando-os nesse vetor. Ao fim da determinização de um estado segue-se para o próximo estado desse vetor. Assim, a remoção de estados

inalcançáveis a partir de S já é feita nesse momento.

Tal etapa é feita a partir dos dados presentes no `map ndfa`, sendo que a saída do processo é posta no `map dfa`. Cabe ressaltar que são concatenados aos estados que passaram pelo processo de determinização um 'I', a fim de diferenciar os estados 'A' e 'B' do estado 'AB', por exemplo, pois isso poderia acontecer dada a forma como os estados foram enumerados.

Por sua vez, a função `minimize` possui duas assinaturas. Em uma delas, onde não há parâmetros, é chamada a outra, que recebe uma `string` 'SI' correspondente ao estado inicial. Nesta é percorrido o autômato removendo os estados mortos de forma recursiva. Após o retorno dessa chamada inicial é executada a função `remove`, que percorre o autômato novamente para remover as transições que vão para os estados removidos anteriormente, finalizando assim o processo de minimização. É importante enfatizar que, como já mencionado anteriormente, a eliminação de estados inalcançáveis já ocorre como subproduto da função `makedet`.

A função `fill` foi implementada a fim de preencher com 'x' todas as posições de símbolos não pertencentes ao conjunto de símbolos de todos os alfabetos das linguagens representadas por tokens e gramáticas. Por fim, a função `csv` salva o autômato gerado em um arquivo denominado `automata.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 ao objetivo a qual se propõe.

6. Conclusão

Conclui-se que a aplicação desenvolvida atende ao objetivo ao qual se propõe, consistindo em 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).

No trabalho foi apresentada uma descrição de um possível processo de construção de analisadores léxicos, assim como realizada a exposição de seu planejamento, funcionamento, particularidades e implementação.

Por fim, foram citados e comentados os arquivos utilizados nos testes para a verificação da funcionalidade da aplicação desenvolvida.

Referências

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