Uso de autômatos finitos na análise léxica para identificação de tokens

Samuel Henrique F. Leite, Olávio G. de Almeida, Luiz Eduardo C. da Mota Leite, Douglas Farias Cordeiro, Núbia Rosa da Silva

Resumo Os Autômatos Finitos podem ser utilizados na fase de Análise Léxica (AL) para identificar tokens de uma linguagem de programação, tendo em vista que analisam caractere a caractere da entrada, que é o princípio da AL. Neste artigo, é apresentado um estudo sobre a aplicação de AL na linguagem de programação C, onde são utilizados Autômatos Finitos para determinação de três grupos de *tokens*: palavras-chave, símbolos especiais e operadores. Foi realizado um estudo analítico sobre o uso de Autômato Finito Determinístico (AFD) e Autômato Finito Não-Determinístico (AFND), de modo a embasar a abordagem mais adequado ao problema tratado. A corretude dos resultados alcançados através da estratégia de exploração de AC proposta mostram sua eficácia frente aos trabalhos correlatos apresentados na revisão de escopo.

1 Introdução

Análise Léxica (AL) é a primeira fase do processo de compilação (**shI˙AnExplorationonLexicalAnalysis**). Trata-se de uma importante fase em linguagens que utilizam métodos de implementação baseados em compilação. O analisador léxico tem como objetivo identificar eficientemente os *tokens* de uma determinada linguagem.

Token é uma unidade lógica, na qual pode ser um único caractere ou uma sequência de caracteres. Alguns exemplos de *tokens* são: identificadores, constantes, símbolos de pontuação e outros terminais especificados pela linguagem (**shI˙AnExplorationonLexicalAnalysis**).

Tokens podem ser divididos em seis classes (**shI˙siteDosTokens**): (I) palavras-chave; (II) identificadores; (III) constantes; (VI) conjunto de caracteres; (V) símbolos especiais; e (VI) operado-

Samuel Henrique F. Leite
Instituto de Biotecnologia, Universidade Federal de Goiás, Catalão, Goiás, Brasil.
e-mail: sussai.andressa@outlook.com

Olávio G. de Almeida
Instituto de Biotecnologia, Universidade Federal de Goiás, Catalão, Goiás, Brasil.
e-mail: sussai.andressa@outlook.com

Luiz Eduardo C. da Mota Leite
Instituto de Biotecnologia, Universidade Federal de Goiás, Catalão, Goiás, Brasil.
e-mail: sussai.andressa@outlook.com

Douglas Farias Cordeiro
Faculdade de Informação e Comunicação, Universidade Federal de Goiás, Goiânia, Goiás, Brasil.
e-mail: cordeiro@ufg.br

Núbia Rosa da Silva
Instituto de Biotecnologia, Universidade Federal de Goiás, Catalão, Goiás, Brasil.
e-mail: nubia@ufg.br

Anais do XV Encontro Anual de Ciência da Computação (EnAComp 2020). ISSN: 2178-6992.
Catalão, Goiás, Brasil. 25 a 27 de Novembro de 2020.

res. Neste trabalho foram levados em consideração os três seguintes grupos de *tokens*: palavras-chave, símbolos especiais e por fim, operadores.

A entrada do analisador léxico é o próprio código fonte do programa. Durante o processo, a tabela de símbolos (composta por *tokens*) é armazenada e os *tokens* servem como entrada para a fase de análise sintática. Em Autômatos Finitos (AFs), assim como na AL, o processo analítico é realizado caractere a caractere no código do programa (**shl˙AnExplorationonLexicalAnalysis**), sendo ideais para serem usados durante a fase de AL para identificação de *tokens*.

Os AFs possuem duas principais abordagens: Autômato Finito Determinístico (AFD) e Autômato Finito Não-Determinístico (AFND). AFD é chamado de determinista porque para cada estado e para cada símbolo de entrada é definida apenas uma transição. Por outro lado, no AFND a função de transição retorna um conjunto de estados em vez de um único estado, sendo denominado de não-determinístico devido à escolha de movimentos que podem levar de um estado a outro (**shl˙34**). Com isso, é proposto neste artigo analisar qual abordagem de AF é mais usada no contexto de análise léxica, e implementar um analisador léxico (para a linguagem de programação C) que utilize essa abordagem.

Uma grande variedade de linguagens de programação está disponível, cada uma com sua finalidade. Linguagens com as mais variadas sintaxes, paradigmas, semânticas, etc. No entanto, ao analisar as linguagens disponíveis, pode-se observar que há algumas que são mais populares[1]. A linguagem C pode ser um exemplo disso, sendo uma linguagem imperativa para propósito geral. Trata-se de uma linguagem amplamente utilizada no meio acadêmico e científico por sua alta eficiência, uma vez que executa instruções em um nível mais baixo.

A linguagem C possui *tokens* amplamente conhecidos por programadores. Estes *tokens* foram utilizados como base para criação de outras linguagens, como por exemplo, Java e C++. Dessa forma, os *tokens* da linguagem de programação C foram escolhidos para o contexto deste artigo. Sendo assim, o objetivo deste é implementar um analisador léxico utilizando AF, que seja capaz de reconhecer os *tokens* da linguagem C, se limitando às seguintes classes de *tokens*: palavras-chave, símbolos especiais e operadores. Embora o escopo deste trabalho está voltado à utilização de AFs para reconhecer os *tokens* da linguagem C, eles podem ser utilizados para reconhecer os *tokens* de qualquer outra linguagem de programação que possua uma estrutura rígida.

Na seção 2 descrita a forma de levantamento bibliográfico para o desenvolvimento do mesmo. Na seção 3, foi descrita a metodologia utilizada para desenvolver o AFD proposto e o algoritmo para realizar a AL. Posteriormente são demonstrados os resultados obtidos através deste trabalho (seção 4). Por fim, na seção 5 são realizadas as considerações finais.

2 Trabalhos correlatos

O estudo de AL utilizando AFs já vem sendo realizado há algum tempo. (**shl˙67**) propôs a utilização de AFDs para identificação de *tokens*, onde o foco do trabalho estava em realizar a identificação de *tokens* que poderiam ter tamanhos infinitos. Devido à limitações em termos de processamento computacional, a identificação de *tokens* na altura da publicação do trabalho

[1] O ranking RedMonk, o qual utiliza dados do GitHub Archive e do Stack Overflow para análise do interesse de desenvolvedores em linguagens de programação destaca a linguagem C entre as dez mais utilizadas no mundo (Fonte: *https://redmonk.com/sogrady/2020/07/27/language-rankings-6-20/*).

acabava por se tornar inviável. A proposta feita por **shl˙67** seria utilizar a verificação do símbolo mais à frente e armazenando o anterior para verificar sua validade.

Em **shl˙56** é realizada a AL da linguagem ATLAS para gerar uma linguagem mais compreensiva do programa sem redundâncias na interpretação do mesmo. Para gerar essa linguagem, o autor utiliza o JFlex (**shl˙klein2010jflex**), um analisador léxico que se baseia no uso de AFDs para identificação de *tokens*, sendo considerados como entrada diversos parâmetros sobre a linguagem e realiza a AL.

É retratado também um estudo sobre as formas de se realizar a AL utilizando AFs em **shl˙34**. Neste estudo é realizada a análise da implementação de AL baseada tanto AFD quanto em AFND. Como todo AFND possui seu AFD equivalente, torna-se mais prático desenvolver o AFD diretamente, assim se evita o processo de utilizar um algoritmo de conversão entre eles.

Já no trabalho descrito em **shl˙1** é proposta uma aplicação para auxiliar estudantes e desenvolvedores de analisadores léxicos. O objetivo do trabalho foi em fornecer uma interface gráfica para que o usuário possa compreender e desenvolver um AFD para realizar a AL. O algoritmo proposto gera um AFND baseado na entrada obtida através da interface gráfica ou por texto. Com o AFND gerado, é utilizado um algoritmo de conversão do AFND para um AFD equivalente, o qual é utilizado como entrada no JFlex. O foco da pesquisa foi a interface gráfica, para a AL foi utilizado o JFlex.

3 Metodologia

Nesta seção serão apresentados os *tokens* que serão utilizados neste artigo, assim como o AFD criado para decidir se determinado *token* é aceito ou não. Se destaca que a opção do uso da abordagem AFD foi realizada devido à sua menor complexidade em termos de processamento computacional, uma vez que os AFND necessitam ser convertidos para AFDs. Por fim, serão mostrados detalhes da implementação feita.

3.1 Tokens

Para que seja possível criar o AFD que é utilizado neste trabalho, primeiramente deve-se identificar quais serão os *tokens* presentes em cada uma das três classes de *tokens* da linguagem C que serão usadas neste artigo. Na Figura 1 são mostrados os *tokens* presentes na classe de palavras-chave.

Totalizando os trinta e dois *tokens* reconhecidos pela linguagem C, estes serão mais tarde identificados pelo analisador léxico, através do AFD que será apresentado. A Figura 2 apresenta os *tokens* pertencentes à classe de operadores e símbolos especiais, que também farão parte do AFD. Vale ressaltar que nem todos os símbolos especiais e operadores suportados pela linguagem C serão considerados no contexto deste trabalho.

Cada operador e símbolo especial descrito na Figura 2 possui sua finalidade na linguagem C. Vale lembrar que alguns operadores descritos podem ser caracterizados também como símbolos especiais. Um exemplo disto é o símbolo asterisco (*), que dependendo do contexto em que se encontra, pode ter duas classificações: operador, que no contexto de uma expressão indica uma

```
auto      double    int       struct
break     else      long      switch
case      enum      register  typedef
char      extern    return    union
const     float     short     unsigned
continue  for       signed    void
default   goto      sizeof    volatile
do        if        static    while
```

Figura 1: Lista de palavras-chave
Fonte: **shl˙siteDosTokens**.

```
Símbolos especiais          Operadores
    [     ]           +  ++  -  --  *  /
    (     )           =  ==  &  &&  |  ||
    {     }           %  >>  >  >=  <  <<
    ,     ;           ?  <=  :  !=  !
```

Figura 2: Lista de símbolos especiais e operadores
Fonte: elaborado pelos autores.

operação de multiplicação de termos; símbolo especial, utilizado para criar ou manipular variáveis do tipo ponteiro.

Outro ponto importante a ser levado em consideração é o tipo do operador. Neste trabalho não é considerado se o mesmo está sendo utilizado de forma errônea no código de entrada do analisador léxico. Como por exemplo, não é verificado se o operador ternário (? :), por exemplo, está de fato operando sobre três termos. No entanto, a solução desenvolvida deverá realizar uma verificação sobre a corretude sintática do código de entrada.

3.2 O AFD

Tendo identificado todos os *tokens* que serão utilizados neste artigo, o AFD pode ser criado. O automato receberá o *token* como entrada, e analisará o mesmo caractere a caractere. Ao final da entrada, caso o automato tenha parado em algum estado final, indica que esta entrada é reconhecida, e a saída do analisador léxico para este deve ser “aceito”. Caso o *token* termine e o AFD não tenha alcançado um estado final, este então será classificado como “rejeitado” pelo analisador léxico. A Figura 3 mostra o AFD criado para classificar os *tokens*, levando em consideração as palavras-chave, símbolos especiais e operadores da linguagem C.

3.3 Implementação

Neste projeto, a linguagem de programação Python foi utilizada na fase de implementação do analisador léxico. Como toda a parte de entrada e saída do programa foi feita utilizando arqui-

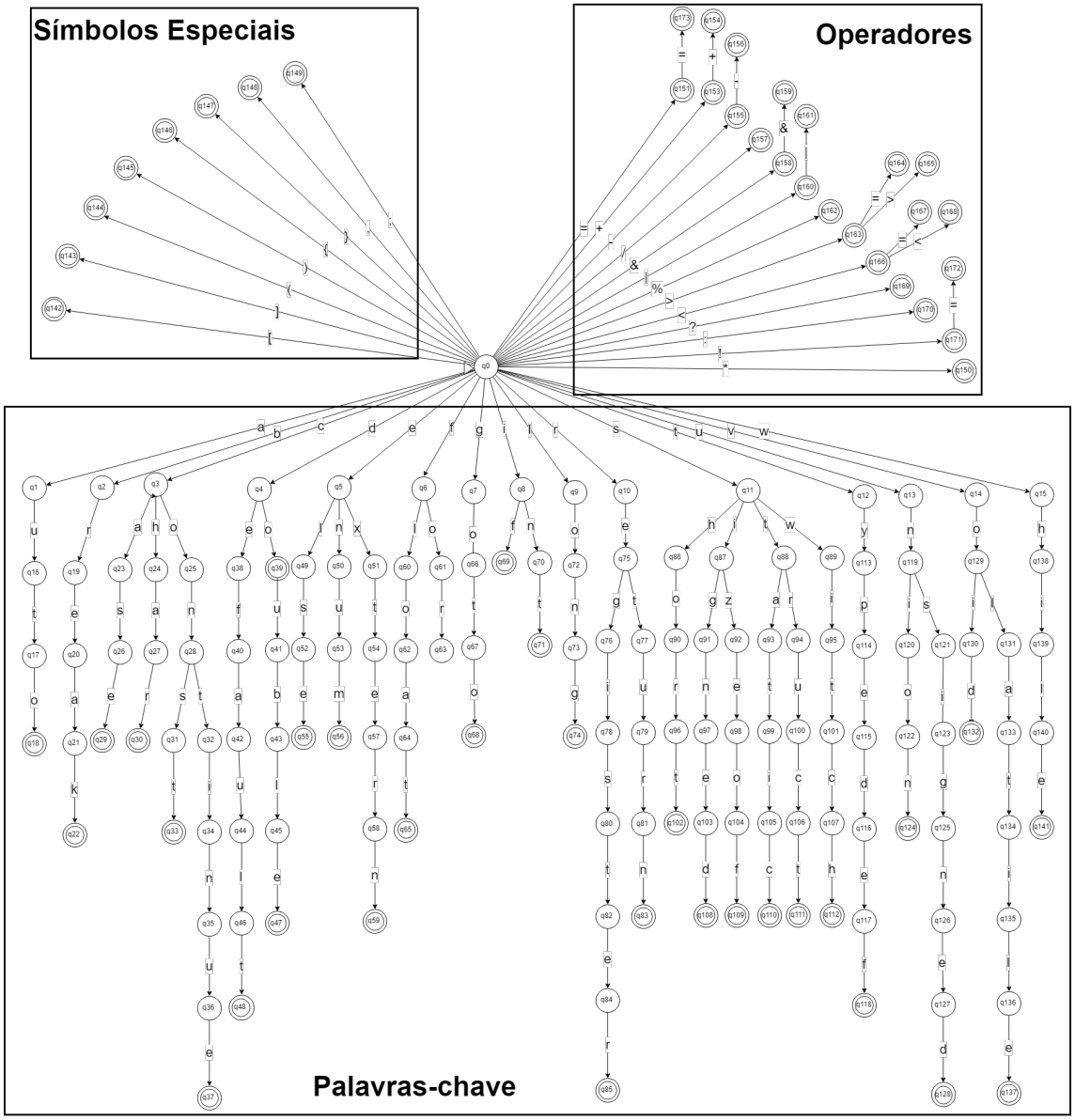

Figura 3: AFD completo.
Fonte: Elaborado pelos autores

vos, a facilidade dessa linguagem de programação para manipulação de arquivos foi um fator importante para a escolha da mesma.

Como dito anteriormente, a entrada e saída do código feito é realizada por meio de arquivos. Sendo assim, foram utilizados três arquivos: o primeiro contendo o código de entrada (código divido em *tokens*); o segundo contendo uma descrição do AFD criado; e por fim, o último contendo a saída do analisador léxico, indicando quais foram os *tokens* aceitos e quais foram rejeitados.

O arquivo contendo o automato foi feito da seguinte maneira: o estado inicial do AFD é situado na primeira linha do arquivo, e está marcado com um símbolo ($>$) antes do estado. Em seguida, na segunda linha estão representados o conjunto de estados finais do autômato. Sendo assim, no código os estados finais serão representados por um vetor de estados. Por fim, o restante do arquivo é divido em três colunas: a primeira representa o estado atual; a segunda coluna

representa o caractere a ser processado; a terceira e última coluna representa o estado seguinte para aquela dada entrada.

A Figura 4 mostra um exemplo de autômato feito para aceitar a palavra-chave 'auto'. O autômato mostra o formato em que o mesmo deve se encontrar no arquivo. Para que este possa reconhecer novas cadeias de caracteres, basta adicionar os novos estados de forma correta no arquivo.

```
1    >q0
2    q4
3    q0 a q1
4    q1 u q2
5    q2 t q3
6    q3 o q4
```

Figura 4: Representação do automato no arquivo.
Fonte: elaborado pelos autores.

Levando em consideração os requisitos referentes à configuração dos arquivos de manipulação da solução proposta, uma implementação do analisador léxico foi feita. A implementação consiste, resumidamente, no seguintes passos: (I) os *tokens* são identificados no arquivo de entrada; (II) cada *token* identificado é passado para o AFD, que decide se aceita ou não. O Algoritmo 3 apresenta pseudocódigo da implementação.

4 Resultados e Experimentos

A primeira parte da implementação tem o objetivo de receber o código de entrada, e dividir a mesma em *tokens*. O código de entrada deve constar em um arquivo, em uma só linha. Um exemplo de código de entrada pode ser visto na Figura 5

```
1    int main(){int a, b = 1; c = 2; a = b + c}
```

Figura 5: Exemplo de entrada para o programa.
Fonte: elaborado pelos autores.

A partir da identificação dos *tokens* presentes no arquivo de entrada, é criado então um novo arquivo, onde os *tokens* serão inseridos. Cada linha do novo arquivo possui um *token*. A Figura 6 mostra como deve ficar este arquivo.

Com os *tokens* separados, basta executar passar cada *token* pelo AFD, que realiza a análise caractere a caractere, decidindo se o mesmo aceita ou não determinada entrada. Um exemplo de saída é apresentado na Figura 7, onde a coluna da esquerda representa o *token*, e na coluna da direita o seu estado (aceito ou rejeitado).

Algoritmo 3: PSEUDOCÓDIGO DA IMPLEMENTAÇÃO

```
1  início
2      D = string do arquivo de entrada;
3      A = automato que aceite os tokens desejados;
4      enquanto ainda houver caractere a ser lido em D faça
5          se o caractere atual é um simbolo então
6              se existem 2 símbolos consecutivos iguais e pertencentes ao conjunto
                [>, <, =, −, &, |, +] então
7                  adicione o token formado pelos dois símbolos no vetor de tokens;
8              senão
9                  se o caractere atual está contido em [>, <, !] e o próximo símbolo é = então
10                     adicione o token formado pelos dois símbolos no vetor de tokens;
11                 senão
12                     adicione o token formado pelo símbolo no vetor de tokens;
13                 fim
14             fim
15         senão
16             enquanto a cadeia de caracteres for formada por letra faça
17                 adicione o caractere ao token;
18             fim
19             adicione o token ao vetor de tokens;
20         fim
21     fim
22     para cada token T no vetor de tokens faça
23         verifique se T é aceito por A;
24     fim
25 fim
```

1	int	10	,
2	main	11	c
3	(	12	;
4	)	13	a
5	{	14	=
6	int	15	b
7	a	16	+
8	,	17	c
9	b	18	}

Figura 6: *Tokens* separados por linha.
Fonte: elaborado pelos autores.

5 Conclusões

A utilização do algoritmo proposto possibilitou a AL (especificamente dos *tokens*: palavras-chave, símbolos especiais e operadores) da linguagem C utilizando AFD.

Este artigo teve como objetivo não só fazer um levantamento bibliográfico, como também uma demonstração de como AFs podem ser utilizados na fase de AL (primeira parte no processo de

1	int	aceito	10	,	aceito
2	main	rejeitado	11	c	rejeitado
3	(	aceito	12	;	aceito
4	)	aceito	13	a	rejeitado
5	{	aceito	14	=	aceito
6	int	aceito	15	b	rejeitado
7	a	rejeitado	16	+	aceito
8	,	aceito	17	c	rejeitado
9	b	rejeitado	18	}	aceito

Figura 7: Saída final para o exemplo de entrada.
Fonte: elaborado pelos autores.

compilação). Analisando a literatura, foi possível perceber que a maioria dos trabalhos encontrados utilizam AFD para cumprir esta tarefa, isso se dá devido a AFNDs necessitarem de ser convertidos para AFDs.

Como principal resultado, observou-se que os três grupos de *tokens* da linguagem C definidos para o trabalho (palavras-chave, símbolos especiais e operadores) puderam ser identificados através do uso de AFs. Sendo assim, pode-se afirmar que AFDs podem ser utilizados de forma eficaz na identificação de *tokens* da linguagem C, e eventualmente de outra linguagem de programação.

Referências

ANTONOPOULOS, Andreas M. **Mastering Bitcoin: unlocking digital cryptocurrencies**. [S.l.]: ”O’Reilly Media, Inc.”, 2014.

BARBER, Simon et al. Bitter to better—how to make bitcoin a better currency. In: SPRINGER. INTERNATIONAL Conference on Financial Cryptography and Data Security. [S.l.: s.n.], 2012. p. 399–414.

BONNEAU, Joseph et al. Anonymity for Bitcoin with accountable mixes. **Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)**, v. 8437, p. 486–504, 2014. ISSN 16113349. DOI: 10.1007/978-3-662-45472-5_31. arXiv: arXiv:1311.0243.

ELDREDGE, Nate. **What is a stealth address?** [S.l.: s.n.], 2014. Disponível em: ¡https://bitcoin.stackexchange.com/questions/20701/what-is-a-stealth-address¿.

IOTEX. **Blockchain Privacy-Enhancing Technology Series - Stealth Address (I)**. [S.l.: s.n.], 2018. Disponível em: ¡https://hackernoon.com/blockchain-privacy-enhancing-technology-series-stealth-address-i-c8a3eb4e4e43¿.

MAXWELL, Gregory. **CoinJoin: Bitcoin privacy for the real world**. [S.l.: s.n.], 2013. Disponível em: ¡https://bitcointalk.org/index.php?topic=279249.0¿.

_____. **CoinSwap: Transaction graph disjoint trustless trading**. [S.l.: s.n.], 2013. Disponível em: ¡https://bitcointalk.org/index.php?topic=321228.0¿.

MEIKLEJOHN, Sarah et al. A fistful of Bitcoins: Characterizing payments among men with no names. In: 6. PROCEEDINGS of the Internet Measurement Conference - IMC ’13. [S.l.: s.n.], 2013. p. 127–140. ISBN 9781450319539. DOI: 10.1145/2504730.2504747. Disponível em: ¡http://dl.acm.org/citation.cfm?id=2504730.2504747¿.

MÖSER, Malte; BÖHME, Rainer. Anonymous alone? measuring Bitcoin’s second-generation anonymization techniques. In: IEEE. 2017 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW). [S.l.: s.n.], 2017. p. 32–41.

NAKAMOTO, Satoshi. **Bitcoin: A peer-to-peer electronic cash system**. [S.l.: s.n.], 2009. Disponível em: ¡http://www.bitcoin.org/bitcoin.pdf¿.

OBER, Micha; KATZENBEISSER, Stefan; HAMACHER, Kay. Structure and anonymity of the bitcoin transaction graph. **Future Internet**, v. 5, n. 2, p. 237–250, 2013. DOI: 10.3390/?5020237.

TODD, Peter. **[Bitcoin-development] Stealth Addresses**. [S.l.: s.n.], 2014. Disponível em: ¡https://lists.linuxfoundation.org/pipermail/bitcoin-dev/2014-January/004020.html¿.

WRIGHT, Thomas. Security, privacy, and anonymity. **XRDS: Crossroads, The ACM Magazine for Students**, v. 11, n. 2, p. 5–5, 2004. ISSN 1528-4972. DOI: 10.1145/1144403.1144408.