

IDENTIFICAÇÃO DE ERROS NA LISTA DE CONECTIVIDADE DE CIRCUITO ELÉTRICO EMPREGANDO ANALISADOR LÉXICO E SINTÁTICO

Sousa, G. T. A.¹, Machado, B. S.², Brodbeck, L. C.³, Galvão Filho, A.R.^{1,2}, Carvalho,
R.V.^{1,2}, Coelho, C.J.^{1,2}

¹ Laboratório de Computação Científica - Escola de Ciências Exatas e da Computação

Pontifícia Universidade Católica de Goiás

Goiânia-Goiás-Brasil

² Mestrado em Engenharia de Produção e Sistemas

Pontifícia Universidade Católica de Goiás

Goiânia-Goiás-Brasil

³ Núcleo de Desenvolvimento de Software

Fundação Aroeira, São Paulo-São Paulo-Brasil

RESUMO: O objetivo deste trabalho é descrever o processo de implementação de um parser para a identificação de erros na lista de conectividade de circuito elétrico para o módulo do sistema de fluxo de potência no contexto de um Gêmeo Digital, empregando analisador léxico e analisador sintático. A metodologia para a implementar o analisador léxico é baseada em linguagem regular produzida a partir de gramática regular. O desenvolvimento do analisador sintático utiliza uma gramática livre de contexto. Os resultados obtidos com a aplicação do parser implementado demonstra que eficácia para a identificação de erros na lista de conectividade do circuito elétrico é de 99,54%. Conclui-se que o parser implementado será útil para a correção de listas de conectividade de circuitos elétricos.

Palavras-chave: Parser, Circuito elétrico, Fluxo de potência.

ABSTRACT: The objective of this paper is to describe the process of implementing a parser for the identification of errors in the electrical circuit netlist for the power flow module in the context of a Digital Twin, using lexical analyzer and syntactic analyzer. The methodology for implementing the lexical parser is based on regular language produced from regular grammar. The development of the parser uses context-free grammar. The results obtained with the application of the implemented parser show that the efficiency for identifying errors in the electrical circuit connectivity list is 99.54%. It is concluded that the implemented parser will be useful for the correction of electrical circuit connectivity lists.

Keywords: *Parsing, Electrical circuit, Power flow*

1. Introdução

O desenvolvimento de um gêmeo digital (*Digital Twin*, DT) ou cópia virtual de usinas hidrelétricas envolve a implementação em software de modelos de circuitos elétricos que necessitam de métodos precisos de avaliação do desempenho do circuito [1]. Devido à complexidade dos circuitos integrados atuais, a análise do circuito auxiliada por computador é ferramenta essencial para obter informações sobre o seu desempenho. Obter o desempenho de um circuito a partir de um protótipo de laboratório é quase impossível [2].

Os circuitos elétricos do Gêmeo Digital são descritos através de uma linguagem de descrição de circuitos e de componentes com terminais conectados a determinadas nós, linguagem derivada do software SPICE (Simulation Program with Integrated Circuit Emphasis). Os grupos de componentes conectados aos nós são chamados de netlists [4]. A descrição dos componentes eletrônicos e sua conectividade no circuito descrito na netlist possui a descrição dos dispositivos/componentes usados nos circuitos. O SPICE recebe a descrição do circuito a ser simulado como um arquivo tipo texto que contém a lista dos elementos do circuito (ramos do circuito) e seus respectivos nós de ligação [5].

O arquivo conhecido como *netlist* pode ser criado manualmente pelo usuário, a partir de algumas regras de sintaxe simples ou gerado de modo automático através de um sistema de desenho do circuito. A criação do arquivo para a descrição do circuito deve observar regras de formais para formação de linguagens [6]. Todos os nós do circuito devem ser numerados incluindo um nó zero (referência à terra) que serve com referência para as tensões calculadas. As versões mais modernas do SPICE identificam os nós por nomes e não apenas números. Cada elemento do circuito deve ter um identificador único [7]. Dessa forma, a primeira letra do nome especifica o tipo de elemento, R indica um resistor, C indica um capacitor e assim por diante, para outros identificadores [8].

A *netlist* construída manualmente, como no caso do DT, está sujeita a erros e depende de muito trabalho manual e muita atenção aos detalhes do circuito elétrico que está sendo descrito para sua devida correção. O trabalho de correção de uma *netlist* é semelhante ao trabalho de um compilador de linguagens de programação, porém, diferentemente da linguagem de programação, a *netlist* tem suas especificidades [9]. Diante a identificação de inconsistências na *netlist*, torna-se necessário o desenvolvimento de um analisador léxico e

sintático. O analisador léxico sintático desenvolvido será incorporado ao DT ora em desenvolvimento.

A análise léxica (*scanning*) consiste no processo de leitura da sequência de caracteres, realizado da esquerda para a direita, que compõem um programa, e também no agrupamento dos caracteres em componentes léxicos (*tokens*) [10]. O componente léxico se refere a uma sequência de caracteres com significado coerente com a linguagem em questão. Uma linguagem de programação apresenta um pequeno número de componentes léxicos (caractere, cadeia de caractere (*string*), inteiro, etc.). O objetivo da análise léxica é facilitar o trabalho realizado na análise sintática. A análise léxica pode ser parte integrante da análise sintática. Porém, existem motivações para manter a análise léxica separada da análise sintática como eficiência, modularidade e tradição [11].

A análise sintática (*parsing*) se refere ao mapeamento de uma sentença para uma representação formal da sua estrutura sintática. A estrutura sintática de uma frase oferece pistas sobre o significado da frase e facilita construir tais estruturas a partir da descrição formal das características sintáticas das palavras (o resultado de uma análise anterior da forma da palavra). A descrição da estrutura sintática deve seguir um modelo de gramática de dependência [12]. Um ponto central do modelo, em relação à análise, são os pacotes. Os pacotes representam o conhecimento sintático de uma língua que é a base para as sentenças da análise da linguagem. Os pacotes são de uma forma muito generalizada, partes de árvores.

Uma das etapas da análise sintática é verificar se os componentes léxicos (*tokens*) gerados na etapa de análise léxica formam uma expressão com significado. Essa operação utiliza a gramática livre de contexto, a qual define um procedimento algorítmico para os componentes. O objetivo é a formação de uma expressão para definir a ordem específica em que os componentes léxicos devem ser colocados em termos da compreensão da linguagem [13].

A análise se refere à quebra de texto comum. Os mecanismos de pesquisa geralmente analisam frases de pesquisa inseridas pelos usuários para que possam pesquisar com mais precisão cada palavra. Alguns programas podem analisar documentos de texto e extrair informações, como nomes ou endereços. Os programas de planilhas podem transformar documentos formatados em tabelas com linhas e colunas analisando o texto [14]. A tecnologia

da análise sintática tem como objetivo decompor automaticamente as estruturas complexas em suas partes. A questão que envolve o desenvolvimento de tecnologias de análise sintática está relacionada com uma grande variedade de aspectos computacionais, linguísticas e matemáticos. Além de questões técnicas relacionadas a recursos de engenharia de software e projeto de sistemas. Alguns dos mais importantes desses problemas são escopo, efetividade, eficiência robustez, linearização e integração [15].

Os algoritmos para análise sintática são projetados para classes gramaticais e não para gramáticas individuais. Os algoritmos devem ser corretos em relação às gramáticas e não devem produzir como resultados da análise uma frase que não possa surgir da gramática em questão. Na maioria dos casos, a análise sintática não é um objetivo em si mesmo. Em geral, a análise sintática é um instrumento para o reconhecimento, interpretação ou transformação de estruturas complexas e se integra a uma estrutura como um compilador, por exemplo. A importância dos problemas relacionados à tecnologia de análise depende muito da aplicação pretendida [16].

Neste trabalho, a análise léxica e a análise sintática serão usada para fazer a correção de termos provenientes de uma *netlist*, construída para circuitos implementados em software no sistema eletrônico de potência do sistema DT. Os módulos da análise léxica e análise sintática serão agrupados em um único programa denominado parser. O parser será implementado em linguagem C++ para ser integrado ao sistema DT.

2. Materiais e Métodos

Na implementação de DT para o controle da operação da Usina Hidrelétrica de Jirau instalada no Rio Madeira no Estado de Rondônia é necessário fazer a validação do arquivo *netlist* que descreve circuitos elétricos através de um parser. As regras para a descrição do circuito *netlist* segue as mesmas regras para a construção de uma *netlist*. Todos os nós do circuito devem receber um número. A presença de um nó zero ou nó terra é obrigatória. O nó zero é a referência para o cálculo das tensões do circuito. A lista *netlist* admite que o nó receba também um nome e não só o número. Cada elemento do circuito deve ter nome único. A primeira letra do nome define o tipo do elemento do circuito. A sintaxe para a descrição dos elementos do circuito tem a forma geral <nome-do-elemento> <nó+> <nó-> <valor>.

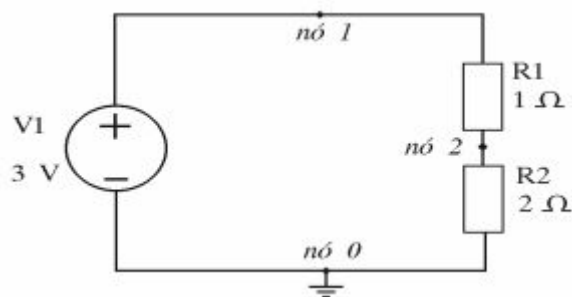


Figura 1: Circuito elétrico ilustrativo.

Tabela 1. Descrição do circuito (*netlist*) ilustrativo da Figura 1.

1	Figura 1. Circuito elétrico ilustrativo	Primeira linha contém o título ou descrição do circuito. Não é usado como linha de comando e é ignorada.
2	V1 1 0 3.0	Fonte de tensão de volts, polo positivo ligado no nó 1, pólo negativo ligado no nó 0.
3	R1 1 2 1.0	Resistor R1, 1 Ω , ligado entre os nós 1 e 2.
4	R2 2 0 2.0	Resistor R2, 2 Ω , ligado entre os nós 2 e 0.

A *netlist* para o circuito da Figura 1, adaptado do trabalho de Mehl [17], contém todos os detalhes do circuito a ser simulado. O nó 0 é o terra do circuito. Os outros dois nós receberam os números 1 e 2, respectivamente. A Tabela 1 descreve todos os elementos do circuito e mostra o conteúdo completo da *netlist* para o circuito da Figura 1.

Para a construção do parser, inicialmente é desenvolvido o módulo de análise léxica. Um arquivo similar ao mostrado na Tabela 1 é lido pelo analisador léxico linha a linha. O analisador léxico representa a fase inicial de implementação do parser para verificar a validade da *netlist* com a descrição dos circuitos elétricos. A análise léxica lê o arquivo com a descrição do circuito e identifica os componentes léxicos ou *tokens* correspondentes e relata os erros encontrados. Por simplicidade refere-se a *tokens* e não a componentes léxicos a partir desse ponto no texto.

Token é uma cadeia de caracteres (sequência do símbolo fundamental caractere) que tem um significado coerente com a linguagem de programação ou a linguagem que se deseja analisar, como a linguagem de representação do circuitos constante da *netlist*. Exemplos de

tokens em linguagem de programação são identificadores, palavras reservadas, símbolos especiais e números. No caso da expressão aritmética, $x := y * 2$; típica da linguagem de programação Pascal, as cadeias x e y são identificadores, o símbolo $:=$ significa atribuição de valor a uma variável, o símbolo $*$ significa multiplicação, 2 é um número. No caso da linguagem de descrição de circuitos, pode-se, analogicamente, definir um identificador ou nome do elemento, nó+, nó- e valor. Assim, a expressão *netlist* R1 1 0 3.0, linha 3 da Tabela 1, produz os *tokens*: R1 <identificador>, 1 <nó+>, 2 e <no->. A linguagem da *netlist* é simples e facilita a implementação do módulo da análise léxica.

O projeto do analisador léxico requer o uso de notações formais para especificar e reconhecer a estrutura dos *tokens* que serão produzidas como saída do analisador. O formalismo, geralmente baseado em expressões regulares e autômatos finitos, evita erros e permite o mapeamento consistente com a linguagem de entrada.

A linguagem formal pode ser expressa a partir de expressões regulares. A gramática regular gera uma expressão regular empregando as operações de concatenação (união de duas cadeias de caracteres), união (de dois ou mais conjuntos) e fecho de Kleene (operação usada para caracterizar certos tipos de autômatos) sobre os elementos de um alfabeto [18, 19].

Um autômato é um modelo abstrato de computador que possui um mecanismo para ler a cadeia de caracteres da entrada baseada em um determinado alfabeto, como a *netlist*. A entrada é escrita em um arquivo de entrada e lida pelo autômato. O autômato não faz modificações no arquivo de entrada. Um autômato determinístico, a cada transição, aceita ou rejeita a cadeia de entrada produzindo um ramo único para cada cadeia de entrada [21, 22].

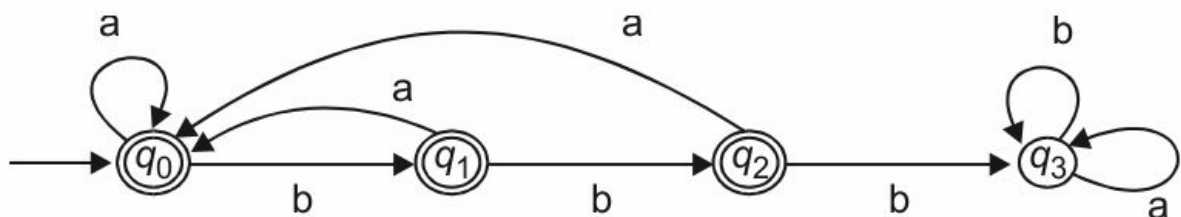


Figura 2: Autômato finito determinístico.

A Figura 2. mostra um Autômato Finito Determinístico (AFD) que aceita a linguagem composta pelos caracteres a e b. A linguagem não pode ter três b seguidos, caso contrário, a

linguagem é rejeitada pelo AFD. Os círculos duplos na Figura 2 representam estados de aceitação da linguagem. Ao atingir um estado de aceitação o AFD finaliza seu ciclo [23]. A implementação do analisador léxico para a análise da *netlist* faz uso de um AFD para produzir os *tokens* da linguagem para representação do circuito elétrico.

O analisador sintático recebe uma sequência de *tokens* produzidos pelo analisador léxico e determina se a cadeia de caractere pode ser gerada através da gramática da linguagem fonte. Caso contrário, o analisador sintático deve reportar erros. O processo da análise sintática é suportado por linguagem formais. Uma linguagem formal se refere ao estudo de modelos matemáticos para a especificação e reconhecimento de linguagens. Em geral, o formalismo usado para análise sintática de linguagens simples, como as linguagens de programação e a linguagem de especificação de circuito elétrico, são as linguagens livres de contexto [24].

Uma linguagem livre de contexto (*Context-Free Languages*, CFL) é uma linguagem gerada por uma gramática livre de contexto (*Context-Free Grammar*, CFG). Uma CFG é uma gramática $G=(V, T, P, S)$, onde V é um conjunto de variáveis da gramática, T é conjunto de terminais, P são as regras de produção da gramática e S é a variável inicial. A CFL determinística é uma classe de linguagem não ambígua e é um subconjunto das CFL. A construção de analisadores sintáticos para linguagens de programação e do tipo *netlist* utiliza a CFL determinística [25, 26].

3. Construção do Parser

No contexto do DT, o parser será usado para detectar erros na *netlist* durante a carga da configuração da descrição de um circuito elétrico do fluxo de potência do Gêmeo Digital, de modo que erros de edição da *netlist* possam ser detectados em tempo de inicialização, contribuindo para a inicialização íntegra do sistema.

O parser construído extrai a informação necessária de uma *netlist* para avaliar sua estrutura. O parser deve ser capaz de montar uma árvore de parse, como mostra a Figura 3, caso contrário, não é possível garantir a correção da tradução dos termos. Uma *netlist* é uma lista de declarações e linhas de controle. Uma declaração do tipo *netlist* define os componentes do circuito, parâmetros do modelos usados para fazer a simulação do circuito ou

definições de sub-circuito. Os elementos da *netlist* ficam claros quando se observa a árvore de parse mostrada na Figura 3.

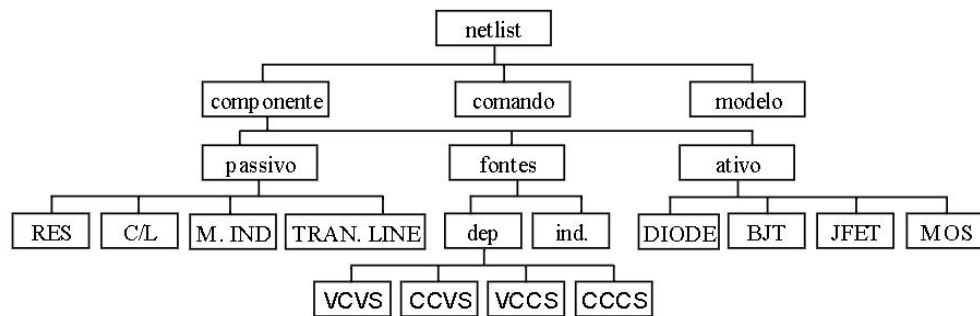


Figura 3. Árvore de parse.

Na árvore da Figura 3, a raiz é representada pela *netlist* e tem como filhos as duas principais categorias de verificação, que são as listas de declaração, representadas pelos componentes e as linhas de controle, representadas pelos comandos e modelos. Os componentes representam a parte mais complexa da árvore, uma vez que deve conter todos os componentes elétricos (ativos, passivos e fontes). A árvore da Figura 3 é fundamental para a construção do parser para a linguagem da *netlist*.

```

01. TITLE UH JIRAU POWER FLOW (March/2018)
02. .subckt COUPLING_525K (1 3)
03. *1,2: primary nodes
04. *2,3: secondary nodes
05. *13.8kV to 525kV
06. Lsecondary (0 3) 2.894612K
07. Lprimary (0 1) 2.
08. K Lsecondary Lprimary 0.9999999
09. .ends COUPLING_525K
10. R_D52A1_6 (166 167) 100.n
11. I_UG01 (8 110) COMPLEX(7.53066K, 7.53066K)
12. .subckt MONOPHASIC_TRANSFORMER_525K (1 2 3)
13. *1,0: primary A
14. *2,0: primary B
15. *3,0: secondary
16. XTEa (1 3) COUPLING_525K
17. XTEb (2 3) COUPLING_525K
18. .ends MONOPHASIC_TRANSFORMER_525K
19. .PRINTDC V(2) I(VSRC) V(23, 17)
20. .END

```

Figura 4: Exemplo de *netlist* a ser analisada.

Toda linguagem possui regras que prescrevem a estrutura sintática de programas bem formados como a CFL. A *netlist* da Figura 4 é um exemplo da *netlist* utilizada para o teste do parser implementado. A seguir a CFG usada para implementar o parser é definida. O conjunto de *tokens* são símbolos terminais (T) básicos que formam as listas de declarações e linhas de controle. Na linha 10 da Figura 4, por exemplo, os terminais são: <R_D52A1_6>; <166>; <167>; <100.n>. E equivalem respectivamente a: <res>; <inteiro>; <inteiro>; <real>. O conjunto de não-terminais (V) é representado pelo conjunto de cadeias de caracteres que definem a linguagem gerada pela gramática. Os símbolos não terminais impõem uma estrutura hierárquica que auxiliam a análise sintática.

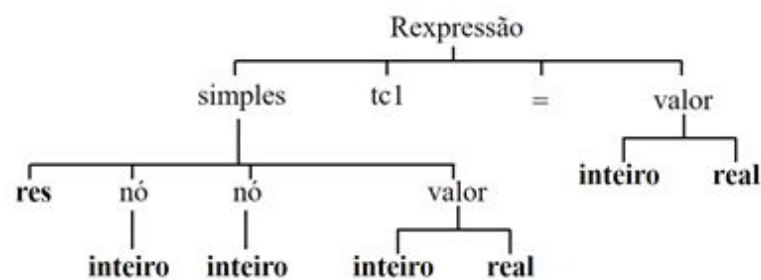


Figura 5: Estrutura hierárquica de símbolos não-terminais.

O conjunto de produções (P) consiste de um não-terminal, chamado de lado esquerdo da produção, que leva a uma sequência de *tokens* e/ou outro não terminal (chamado de lado direito da produção).

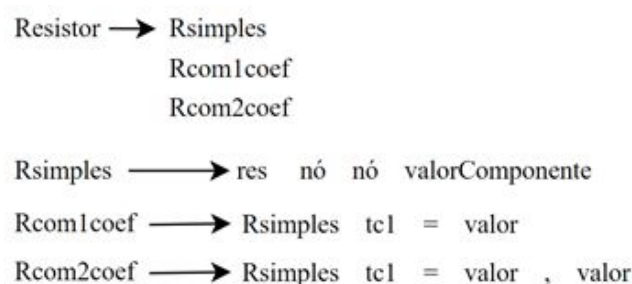


Figura 6: Produções para declarações de resistores.

O símbolo inicial não-terminal, é gerado a partir do primeiro caractere do primeiro *token*. Como na linha 3 da Tabela 1, a declaração R1 1 2 1.0 onde o primeiro caractere do primeiro *token* R1 é a letra R, portanto levando a um não-terminal inicial de Resistor o que leva a uma produção para resistor como mostra a Figura 6.[26]

A estrutura do parser implementado é dividida em duas partes: a análise léxica e a análise sintática. A análise léxica agrupa os caracteres do arquivo de entrada (que contém a descrição do circuito) em *tokens*. Um *token* da *netlist*, é similar a uma palavra da *netlist* usada no SPICE, é composto por duas partes principais: um tipo e um valor opcional. O tipo indica que espécie de palavra o *token* representa. Um número, um sinal de pontuação, um identificador (nome de variável ou função), etc. O valor é usado em alguns tipos de *tokens* para armazenar alguma informação adicional necessária.

Outras informações podem ser associadas a cada *token*, dependendo das necessidades da análise léxica. Um exemplo comum é a posição no arquivo de entrada (linha e coluna) onde o *token* começa para auxiliar o tratamento de erros. No parser implementado, os *tokens* possuem diferentes valores que podem variar dependendo da sua posição ou quantidade total deles na sentença. São primeiramente separados palavra por palavra e então são salvos sequencialmente da seguinte no formato “token” + [Posição].

A descrição de um resistor, por exemplo, ficaria da forma mostrada na Figura 7.

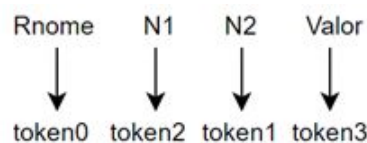


Figura 7: Estrutura dos tokens em uma declaração de resistor.

A cada *token* da Figura 7 é atribuída um valor formado por uma combinação de valor chave e um valor mapeado. Esse mapa contendo todos os *tokens* da sentença como mostra a Figura 6 são passados como parâmetro quando o programa principal chama o módulo da análise sintática.

A tarefa do analisador sintático é simples de realizar quando parte-se dos *tokens* da entrada e não de caracteres isolados. A comunicação entre os analisadores léxico e sintático é sequencial. O analisador léxico produz toda a sequência de *tokens* criada a partir da *netlist* de entrada e passa essa sequência inteira para o analisador sintático.

O analisador sintático produz uma árvore de produção a qual possibilita ver como a gramática é aplicada na sentença. Porém, como a gramática do analisador foi desenvolvida a partir de documentação pré-existente é desnecessário produzir uma árvore. Nesse caso, é feita a linearização da árvore de produção. Existem três maneiras de fazer a linearização de uma

árvore: prefixa ou pré-fixada, infixa e pós fixa. Na implementação do parser para a *netlist* é utilizada a prefixa, como mostra a Figura 8.

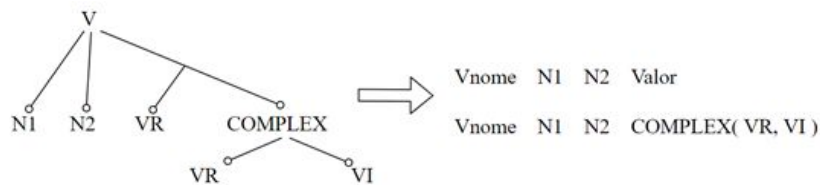


Figura 8: Árvore de produção e sua linearização.

4. Resultados e Discussão

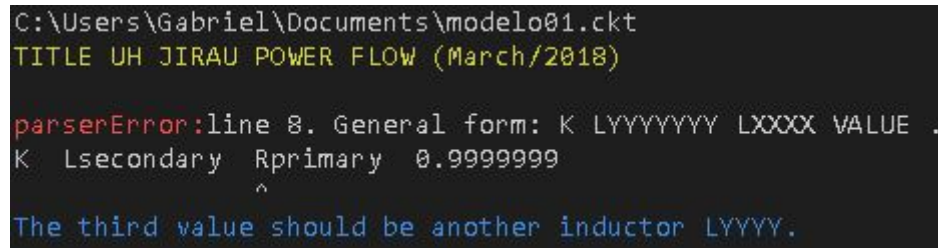
A implementação do analisador sintático teve como objetivo analisar termos da lista de conectividade de circuitos elétricos e a customização de regras da gramática padrão do SPICE. Cada implementação possui seus requisitos. Adaptações para a produção de novas implementações a partir de implementações já existentes requerem a observação de detalhes para que os requisitos legados continuem funcionando harmonicamente na nova implementação. No caso da construção do parser para a *netlist*, foi necessário conciliar o funcionamento de antigos requisitos com novos requisitos. Modificações na gramática do SPICE foram introduzidas para a construção do novo parser.

Para facilitar as customizações realizadas, o parser foi desenvolvido seguindo um regime modular, onde as análises são realizadas a partir de parâmetros iguais em sua maioria. Toda a análise é executada de forma independente uma das outras, o que contribui para o apontamento de erros sem que o programa seja interrompido.

Os módulos podem ser divididos três categorias: valores, declarações e pilhas. Para os valores, foi realizada a análise sintática considerando que estes poderiam ser inteiros ou reais, e também que poderiam ter fatores de escala (como 10kHz). Dessa forma, foram desenvolvidas duas funções, uma para validar inteiros e outra para reais. Estas funções continham todos os sufixos válidos, podendo estes serem retirados ou adicionados facilmente.

Para os módulos de declarações, foi desenvolvida uma função para cada componente, comando ou modelo contido na descrição do circuito. A análise realizada no primeiro caractere do primeiro *token*, direciona o programa para a função correta onde a estrutura é analisada. Por fim, algumas das regras são baseadas em pilhas, sendo a principal delas a

declaração de sub-circuito, que deve conter um marco tanto no início quanto no fim. Ainda, tal sub-circuito pode agir recursivamente como uma pilha, contendo dentro de um sub-circuito outro sub-circuito, ou seja, pode-se ter circuitos aninhados.



```
C:\Users\Gabriel\Documents\modelo01.ckt
TITLE UH JIRAU POWER FLOW (March/2018)

parserError:line 8. General form: K LYYYYYYY LXXXX VALUE .
K Lsecondary Rprimary 0.9999999
      ^
The third value should be another inductor LYYYY.
```

Figura 9: Identificação de erros feita pelo analisador.

A Figura 9 mostra o terminal do computador com os resultados da execução do parser implementado para a *netlist* da Figura 4. A linha 8 da lista foi alterada de uma declaração do segundo indutor <Lprimary> para a declaração de um resistor <Rprimary>. Como a definição de agrupamento de indutores deve ser feita como <K>; <Lindutor1>; <Lindutor2>; e <Valor>, o programa encontra um erro de formatação da linguagem *netlist* e o exibe na tela sinalizando onde o erro ocorre. A mensagem *The third value should be another inductor LYYYY* (O terceiro valor deve ser outro indutor LYYYY) é mostrada para reforçar o que ocorreu. O usuário deve ir na linha indicada, fazer as correções para ter uma nova *netlist* que pode ser usada sem erros para fazer a simulação do circuito.

5. Conclusão

O estudo em questão apresentou as estratégias, métodos e técnicas para a implementação de um parser para a identificação de erros na lista de descrição de circuito descritos em uma *netlist*. A implementação do parser deu ênfase à modularização visando futuras customizações de regras baseadas na gramática livre de contexto. Observou-se ainda, a utilidade da categorização dos módulos individuais de cada regra da gramática que descreve a *netlist*.

Para o processamento de três *netlist* com um total de 7.375 linhas, foram encontrados 34 erros com uma acurácia de 99,54%. A previsão é que depois das correções de erros na implementação a acurácia seja de 100%.

Por fim, algumas considerações para estudos futuros seria aprimorar a indicação de erros feito pelo analisador para facilitar a interação com o usuário. Estudar o desempenho do parser implementado para verificar a necessidade de melhorar seu desempenho. Ainda, uma outra possibilidade seria implementar uma versão paralela do parser empregando a tecnologia de arquitetura de dispositivo de computação unificada (*Compute Unified Device Architecture*, CUDA).

Agradecimentos

Os autores agradecem à Energia Sustentável do Brasil pelo apoio para realização deste estudo projeto ANEEL de código PD-06631-0007/2018 e contrato ESRB 064/2018.

6. Referências Bibliográficas

1. Hmurcik, L., Hettinger, M., Gottschalck, K. S., Fitchen, F. C. SPICE Applications to an Undergraduate Electronics Program, IEEE Transactions on education, v. 33, n. 2, 1990.
2. Rashid, M. H., Rashid, H. M., SPICE for Power Electronics and Electric Power, CRC Press, 2006.
3. Pederson, D. O. A Historical Review of Circuit Simulation, IEEE Transaction on Circuits and systems, v. 3, n. 1, 1984.
4. Tuinenga, P. W. SPICE a guide to circuit simulation and analysis using PSPICE, Prentice Hall, 1988.
5. Vasileska, D., Mamaluy, D., Khan, H. R., Raleva, K., Goodnick, S. M., Semiconductor Device Modeling, Journal of computational and theoretical nanoscience, v. 5, p. 1–32, 2008.
6. Massobrio, G., Antognetti, P., Semiconductor Device Modeling With SPICE, McGraw-Hill, 1993.
7. Tucker, P., SPICE netlist generation for electrical parasitic modeling of multi-chip power module designs, Department of Electrical Engineering College of Engineering University of Arkansas Fayetteville, AR, 2013.
8. Ratier, N., Addouche, M., Gillet., Brendel, R., Parsing Spice Netlists Using a Typed Functional Language, "4th WSEAS International Conference on Mathematical Methods and Computational Techniques in Electrical Engineering, Vouliagmeni, Athens: Grece (2002).
9. Batas, D., Fiedler, H., A Python Interface for SPICE-based Simulations, The international on signal and electronic systems, Poland, 2010.
10. Goyal, A., Yadav, R. Content and lexical analysis: practical application, International Journal of Computer Science and Mobile Computing, v. 3, n.10, p. 356-366, 2014.
11. Mogensen, T. A., Basics of Compiler Design, <http://www.diku.dk/~torbenm/Basics>, acesso 25/10/2019.
12. Kunze, J. Abhängigkeitsgrammatik, American Journal of Computational Linguistics, p. 47-52, 1978.
13. Pulman, S. G., Basic Parsing Techniques: an introductory survey, University of Cambridge Computer Laboratory, and SRI International, Cambridge, 1991.
14. Koopman, H., Sportiche, D., Stabler, E. An Introduction to Syntactic Analysis and Theory, Wiley-Blackwell, 2013.

15. Bunt, H, Tomita, M. Recent Advances in Parsing Technology, Kluwer Academic Publishers, 1996.
16. Lam, M. S., Sethi, R., Ullman, J. D., Compilers: Principles, Techniques, and Tools, Addison Wesley, 2006.
17. Mehl, E. L.de M. Simulação de circuitos eletrônicos em computadores, Universidade Federal do Paraná, Setor de Tecnologia, Curso de Engenharia Elétrica, <http://www.eletrica.ufpr.br/mehl/te236/apostilaPSpice.pdf>, acesso em 28/10/2019.
18. Shallit, J., Review of formal languages and automata theory, Cambridge University Press, <https://pdfs.semanticscholar.org/9f31/415c6d8e4b8bd1f1505411567aa04777b90c.pdf>, acesso em 29/10/2019.
19. Meduna, A. Formal Languages and Computation: Models and Their Applications, CRC Press, 2014.
20. Anuradha, K. Vijayalata, Y., Formal Languages and Automata Theory, CBS Publishers and Distributors PVT LTD, 2019.
21. Xavier, S. P. E.Theory of automata, formal languages and computation, New Age International (P) Ltd., Publishers, 2005.
22. Prithi, S., Sumathi, S., A Review on Deterministic Finite Automata Compression Strategies for Deep Packet Inspection, International Journal of Innovations & Advancement in Computer Science IJIACS, v. 5, n. 6, p. 2347-8616, 2016.
23. Pisolkar, U., Lahane, R., S. A Review on Deterministic Finite Automata Compression Techniques for Efficient Pattern Matching Process, IJCSIT International Journal of Computer Science and Information Technologies, v. 5, n. 6 , p. 7323-7325, 2014.
24. Simovici, D. A., Tenney, R. L., Theory of Formal Languages with Applications, World Scientific Pub Co Inc, 1999.
25. Seymour, G. The Mathematical Theory of Context Free Languages, McGraw-Hill, 1996.
26. Chamlian, V. S. Syntactic Analysis of the SPICE 2G6 Language, McGill University, 1991.