Pós-Graduação em Ciência da Computação

Allison Magno Eugênio da Silva

Implementação de Conversão de Provas $\mathcal{ALC}$ para o Cálculo de Sequentes

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
http://cin.ufpe.br/~posgraduacao

Recife
2019

Allison Magno Eugênio da Silva

Implementação de Conversão de Provas $\mathcal{ALC}$ para o Cálculo de Sequentes

Trabalho apresentado ao Programa de Pós-graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para obtenção do grau de Mestre em Ciência da Computação.

Área de Concentração: Inteligência Computacional
Orientador: Fred Freitas
Coorientadora: Eunice Palmeira da Silva

Recife
2019

Catalogação na fonte
Bibliotecária Monick Raquel Silvestre da S. Portes, CRB4-1217

S586i Silva, Allison Magno Eugênio da
Implementação de conversão de provas ALC para o cálculo de sequentes / Allison Magno Eugênio da Silva. – 2019.
119 f.: il., fig., tab.

Orientador: Frederico Luiz Gonçalves de Freitas.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIn, Ciência da Computação, Recife, 2019.
Inclui referências, apêndices e anexos.

1. Inteligência computacional. 2. Lógica de descrições. I. Freitas, Frederico Luiz Gonçalves de (orientador). II. Título.

006.31 CDD (23. ed.) UFPE - CCEN 2020 - 41

Allison Magno Eugênio da Silva

“Implementação de Conversão de Provas ALC para Linguagem Natural”

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Pernambuco, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Aprovado em: 2 de setembro de 2019.

_____

Orientador: Prof. Dr. Frederico Luiz Gonçalves de Freitas

BANCA EXAMINADORA

_____

Prof. Dr. Fernando Maciano de Paula Neto
Centro de Informática/UFPE

_____

Profa. Dra. Ana Teresa de Castro Martins
Departamento de Computação / UFC

_____

Profa. Dra. Eunice Palmeira da Silva
Instituto Federal de Alagoas – Campus Maceió
(**co-orientadora**)

Em tempo: No título onde se lê: Implementação de Conversão de Provas ALC para Linguagem Natural, leia-se: Implementação de Conversão de Provas $\mathscr{A}\mathscr{L}\mathscr{C}$ para Cálculo de Sequentes.

Decido este trabalho a minha família e a minha esposa que foram porto seguro perante as dificuldades durante este percurso.

AGRADECIMENTOS

A Deus, pela vida.

Aos meus pais, Aparecida e Eugênio, pelo incentivo ao estudo e à vida acadêmica desde minha infância. Pelo apoio incondicional na decisão de sair de Petrolina e morar em Recife. Pelo carinho e afeto que sempre nos mantém unidos.

A minha irmã, Eumara, por sempre acreditar em mim e me proporcionar momentos mais leves na minha vida. Ótimas conversas em inglês por mensagens marcaram esse período em Recife.

A minha esposa, Poliana, por deixar sua família em Petrolina e me acompanhar nesse caminho de mestrado, estando sempre comigo nos momentos legais e nos momentos difíceis nesse período em Recife. Pelo apoio incondicional em todas as horas e em todos os momentos.

A minha tia, Josefa, por me proporcionar momentos familiares e por me acolher tão bem nos primeiros dias/meses em Recife.

A minha família, de modo geral, por sempre me apoiar e me motivar a continuar estudando.

Aos professores da minha graduação, por todo conhecimento compartilhado durante os 5 anos de curso, vocês são incríveis.

Aos professores das disciplinas que cursei no CIn, por todo conhecimento compartilhado durante as aulas.

À Eunice, pelo tempo e total apoio dedicado a me auxiliar no desenvolvimento deste trabalho.

Ao professor Fred, pela confiança em mim depositada para o início, o desenvolvimento e a conclusão deste trabalho.

Aos membros da banca, por aceitarem contribuir com este trabalho.

A todas as pessoas, que direta ou indiretamente, contribuíram ou me apoiaram durante esse período de mestrado, o meu MUITO OBRIGADO!!!

RESUMO

Em raciocínio automático, os usuários necessitam usar os sistemas de inferência não apenas para entender a conclusão resultante desse raciocínio, mas, também para saber como os sistemas chegaram naquelas conclusões, grande parte da legibilidade da prova é perdida para usuários que não possuem o domínio da lógica. O Método de Conexões realiza provas que são consideradas de difícil compreensão, pois, a matriz de prova gerada por esse método possui várias conexões que unem fórmulas atômicas complementares que são verificadas ao percorrer os caminhos da matriz. Este trabalho apresenta uma implementação do método de conversão das provas em $\mathcal{ALC}$ geradas pelo método das conexões para um sistema de sequentes $\mathcal{ALC}$, formalizado por Palmeira (2017). No processo de implementação, são codificadas as etapas propostas na formalização para gerar a prova no cálculo de sequentes em $\mathcal{ALC}$. Com esse processo de conversão, é possível deixar a prova mais legível para usuários comuns, que podem ser detentores do conhecimento do domínio, apenas. O método implementado neste trabalho recebe a fórmula $\mathcal{ALC}$, a correspondente prova de conexões em formato não-clausal e as suas conexões. Uma prova no Cálculo de Sequentes $\mathcal{ALC}$ vai sendo construída e, por fim, é gerada a saída com a prova completa em sequentes. A expressividade da lógica de descrições $\mathcal{ALC}$ é unida ao bom desempenho dos provadores automáticos de teorema, proporcionando uma saída mais amigável e compreensível do raciocínio automático.

Palavras-chaves: Lógica de Descrições. *Attributive Concept Language with Complements* ($\mathcal{ALC}$). Cálculo de Sequentes.

ABSTRACT

In automatic reasoning, users need to use inference systems not only to understand the resultant conclusion of that reasoning, but also to know how the systems came to those conclusions, much of the proof readability is lost to users who do not have the domain of logic. The Connections Method performs tests that are considered difficult to understand because the proof matrix generated by this method has several connections that join complementary atomic formulas that are verified by traversing the paths of the matrix. This paper presents an implementation of the method of converting the $\mathcal{ALC}$ proofs generated by the connections method to a $\mathcal{ALC}$ string system, formalized by **??**). In the implementation process, the proposed formalization steps are coded to generate the $\mathcal{ALC}$ sequence calculation test. With this conversion process, you can make the proof more readable to ordinary users, who may have domain knowledge only. The method implemented in this paper receives the formula $\mathcal{ALC}$, the corresponding proof of non-clause connections and their connections. A Sequence Calculation test $\mathcal{ALC}$ is being built and, finally, the output with the complete sequence test is generated. The expressiveness of the $\mathcal{ALC}$ description logic is coupled with the good performance of automatic theorem provers, providing a little more friendly and understandable output of automatic reasoning.

Keywords: Logic of Descriptions. *Attributive Concept Language with Complements* ($\mathcal{ALC}$). Sequence Calculation.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE SÍMBOLOS

γ	gama
Γ	Gama
Λ	Lambda
$\in$	Pertence
$\exists$	Existe
$\forall$	Para todo
θ	Teta
σ	Sigma
μ	Mi
α	alpha
β	beta
δ	delta
Δ	Delta
θ	theta
Σ	Sigma
τ	tau
φ	fi
ψ	psi

SUMÁRIO

1 INTRODUCAO

A Lógica de Descrições é proposta por Baader et al. (2003). Ela é considerada um subconjunto da Lógica de Primeira Ordem, porém, decidível. Ela é uma família de formalismos que possui ampla utilização na Web Semântica por sua expressividade. Ela oferece um significado inequívoco às descrições de domínio e, com sua semântica bem definida e formal, existem sistemas para raciocínio automático baseados nessa lógica.

O Método de Conexões (BIBEL, 1987) está sendo utilizado cada vez mais por sistemas que visam a finalidade do raciocínio automático, sistemas que podem deduzir automaticamente conhecimento implícito do conhecimento explícito. Por suas vez, as provas geradas por esse método são de difícil compreensão.

Com o uso da Lógica de Descrições e dos sistemas baseados no Método de Conexões, Palmeira (2017) formalizou um método de conversão de provas geradas pelo Método de Conexões para o Cálculo de Sequentes $\mathcal{ALC}$, uma tendência em oferecer esses serviços no processo de apoio à decisão. Com os resultados em provas no cálculo de Sequentes, é possível apresentar os resultados de forma mais amigável, descrevendo como as conclusões foram obtidas.

A motivação para esse trabalho é transformar as provas das conclusões no cálculo de sequentes, implementando o método proposto por Palmeira (2017). Com a codificação do sistema, é possível chegar cada vez mais próximo de uma prova que proporcione um melhor entendimento para o usuário, tornando assim, a prova um pouco mais compreensível por aqueles que não possuem o domínio da lógica. Com os resultados desse trabalho, é possível auxiliar usuários no apoio à decisão, vendo que, o usuário não necessita necessariamente de conhecimentos lógicos para interpretar a saída proposta pela conversão de sua prova.

1.1 ESTRUTURA DA DISSERTAÇÃO

O presente trabalho está dividido nos seguintes capítulos:

- Capítulo 1: Uma breve introdução do que será abortado no decorrer da dissertação.
- Capítulo 2: Fundamentação teórica sobre Lógica de Descrições, Sistemas de Prova, Método de Conexões, Cálculo de Sequentes, Processamento de Linguagem Natural, conceitos esses necessários para o entendimento do tema desse trabalho.
- Capítulo 3: Mostra o processo de conversão proposto por Palmeira (2017), detalhando suas definições e o seu método de conversão formalizado.
- Capítulo 4: Nesse capítulo é apresentado o processo de conversão de provas em lógica de descrições $\mathcal{ALC}$ para o cálculo de sequentes, sistema aqui implementado.

- Capítulo 5: Nesse capítulo é apresentada uma breve conclusão do trabalho implementado, mostrando suas contribuições e os possíveis trabalhos futuros que tomarão esse trabalho como base.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta as principais teorias subjacentes aos assuntos abordados nesta dissertação. Ele introduz brevemente Lógica de Descrições (sub-seção 2.1), Sistemas de Prova (sub-seção 2.2), o Cálculo de Conexões com o Cálculo Clausal e com o Cálculo Não-Clausal para $\mathcal{ALC}$ (sub-seção 2.2.2) e o Cálculo de Sequentes para Subsunção em $\mathcal{ALC}$ (sub-seção 2.3.1).

2.1 LÓGICA DE DESCRIÇÕES

Esta seção apresenta uma introdução sobre Lógica de Descrições e uma definição da sintaxe e da semântica da Lógica de Descrições $\mathcal{ALC}$. Neste trabalho, esse formalismo é utilizado como teoria para o entendimento do Cálculo de Sequentes $\mathcal{ALC}$, incluindo seus conjuntos de construtores básicos (conjunção, disjunção, negação, restrição existencial e restrição universal).

Lógica de Descrições ($DescriptionLogic - DL$) é uma família de formalismos de representação de conhecimento. Com ela, podemos representar conceitos, instâncias, relações entre conceitos e entre conceitos e instâncias. É bastante utilizada na área de inteligência artificial simbólica e é amplamente utilizada para criar modelos de ontologias. Ela é bem sucedida em várias aplicações devido a sua expressividade e é entendida como um subconjunto decidível da lógica de primeira ordem.

Esse formalismo lógico possibilita o raciocínio sobre as bases de conhecimento. Ele oportuniza o poder de expressão e raciocínio, oferecendo apoio para solucionar problemas complexos como a verificação de consistência das leis (FREITAS; JR; STUCKENSCHMIDT, 2011), reutilização forense de análise de padrões de comportamento na vigilância visual (HAN; HUTTER; STECHELE, 2011), representação terminológica médica (NOY et al., 2008) e a importação e reutilização de conceitos múltiplos domínios (BORGIDA; SERAFINI, 2003).

Segundo Ribeiro (2012), é devido ao fato de que as Lógicas de Descrições são fragmentos da lógica de primeira ordem, com a vantagem de serem decidíveis, que elas possuem um potencial de solução de problemas complexos. $\mathcal{ALC}$ é simples, mas, poderosa o suficiente para definir ontologias não triviais, ela constitui a base para outras lógicas de descrições mais expressivas.

2.1.1 Sintaxe e Semântica $\mathcal{ALC}$

Manfred e Smolka (1991) definem $\mathcal{ALC}$ como uma tupla ordenada (N_C, N_R, N_O). Nc representa um conjunto de conceitos, Nr um conjunto de relações e No um conjunto de indivíduos ou constantes, instâncias de N_C e N_R. Essa liguagem contém os seguintes construtores:

- $\sqcap$ (conjunção),
- $\sqcup$ (disjunção),
- $\neg$ (negação),
- $\forall$ (restrição de valor) e
- $\exists$ (restrição existencial)

Abaixo são apresentadas definições de acordo com Baader, Horrocks e Sattler (2008).

Definição 1 *Sintaxe $\mathcal{ALC}$: O conjunto de conceitos $\mathcal{ALC}$ sobre Nc e Nr é definido de tal modo que:*

- $\top$ *(top concept),* $\bot$ *(bottom concept) e cada conceito* $A \in Nc$ *são conceitos* $\mathcal{ALC}$.
- *se* C *e* D *são conceitos* $\mathcal{ALC}$ *e* $R \in N_R$, *então* $C \sqcap D$, $C \sqcup D$, $\neg C$, $\forall R.C$, *e* $\exists R.C$ *são conceitos* $\mathcal{ALC}$.

Definição 2 *Semântica de $\mathcal{ALC}$: Uma interpretção $I=(\Delta^I,.^I)$sobre a tupla (N_C, N_R,N_O) consiste de um conjunto não vazio Δ^I, chamado o domínio de I, e uma função $.^I$ que mapeia*

- *todo conceito* $\mathcal{ALC}$ *a um subconjunto de* Δ^I,
- *e toda relação a subconjunto de* $\Delta^I \times \Delta^I$ *tal que, para todos os conceitos* $\mathcal{ALC}$ C, D *e todas as relações* R, *o mapeamento* $.^I$ *pode ser estendido para conceitos como abaixo.*
- $\top^I = \Delta^I$
- $\bot^I = \emptyset$
- $(C \sqcap D)^I = C^I \cap D^I$
- $(C \sqcup D)^I = C^I \cup D^I$
- $(\neg\ C)^{\ I} = \Delta^I \setminus C^I$
- $(\forall R.C)^I = \left\{x \in \Delta^I \mid \forall_y \in \Delta^i, se(x,y) \in R^I, entãoy \in C^I\right\}$
- $(\exists R.C)^I = \left\{x \in \Delta^I \mid \exists y \in \Delta^I com(x,y) \in r^I, entãoy \in C^I\right\}$

É dito que $C^I(R^I)$ é a extensão do conceito C (da função R) na interpretação I. Se $x \in C^I$, então é dito que x é uma instância de C em I.

2.1.2 Inferências em $\mathcal{ALC}$

Inferências sobre expressões conceituais podem ser definidas como satisfação, subsunção, equivalência e disjunção. Para inferir, são precisos algoritmos que realizem inferências básicas a partir de declarações explícitas na TBox, uma conceituação associada a um conjunto de fatos, e na ABox, um fato associado a um modelo conceitual ou ontologias dentro de uma base de conhecimento. Sendo T uma TBox, abaixo são apresentados todos os tipos de inferências, como em Baader, Horrocks e Sattler (2008).

Definição 3 TBox: *Uma TBox é um conjunto finito de inclusões de conceito geral da forma $C \sqsubseteq D$, onde C, D são conceitos $\mathcal{ALC}$ $C \equiv D$ é uma abreviação para o par simétrico de $C \sqsubseteq D$ e $D \sqsubseteq C$. Uma interpretação $\mathcal{I}$ é um modelo de inclusão de conceito geral $C \sqsubseteq D$ se $C^I \subseteq D^I$. I é um modelo de uma TBox $\mathcal{T}$ se $\mathcal{I}$ é um modelo de cada inclusão de conceito geral em $\mathcal{T}$.*

Definição 4 ABox: *Uma ABox é um conjunto finito de axiomas da forma C(a) ou R(b,c), onde C é um conceito $\mathcal{ALC}$, R é uma função, e a, b e c são indivíduos. Uma interpretação $\mathcal{I}$ é um modelo de um axioma C(a) se $a^I \in C^I$, e $\mathcal{I}$ é um modelo de um axioma R(b,c) se $(b^I, c^I) \in R^I$. $\mathcal{I}$ é um modelo de uma ABox A se $\mathcal{I}$ é um modelo de cada axioma em A.*

Exemplo(TBox/ABox):

Animal $\equiv$ Mamifero $\sqcup$ Passaro (1)
AnimalRiscoExtincao $\equiv$ Animal $\sqcap$ EspecieRiscoExtincao (2)
AnimalSemRiscoExtincao $\equiv$ Animal $\sqcap \neg$AnimalRiscoExtincao (3)
Traficante $\equiv$ Pessoa $\sqcap \exists$trafica.Animal (4)
CriminosoAmbiental $\equiv$ Pessoa $\sqcap \exists$trafica.AnimalRiscoExtincao (5)
Pessoa(JOAO) (6)
AnimalRiscoExtincao(CHITA) $\sqcap$ Mamifero(CHITA) (7)
trafica(JOAO,CHITA) (8)

Em Exemplo(TBox/ABox), Animal, Mamifero, Passaro, AnimalRiscoExtincao, AnimalSemRiscoExtincao, Traficante, Pessoa e CriminosoAmbiental são conceitos; trafica é uma relação, e todos eles são definidos em TBox. Já as declarações (6), (7) e (8) são definidas em ABox.

Definição 5 *Satisfação:Um conceito C é satisfatível com respeito a T se existe um modelo I de T, tal que C^I é não vazio. Neste caso também é dito que I é um modelo de C.*

Exemplo(Satisfação): O conceito AnimalRiscoExtincao $\sqcap$ Mamifero é satisfatível, enquanto o conceito AnimalRiscoExtincao $\sqcap$ AnimalSemRiscoExtincao não é.

Definição 6 *Subsunção:Um conceito C é subsumido por um conceito D com respeito a T se $C^I \subseteq D^I$ para todo modelo I de T. Este caso é denotado por $T \models C \sqsubseteq D$.*

Exemplo(Subsunção) ManterEmCativeiro $\sqsubseteq$ Acao $\sqcap \exists$feitaPor.Pessoa $\sqcap \exists$contra.Animal. Esta subsunção declara que ManterEmCativeiro é uma entre, outras possíveis ações, praticada por pessoas contra os animais.

Definição 7 *Equivalência: Dois conceitos C e D são equivalentes com respeito a T se $C^I = D^I$ para todo modelo I de T. Este caso é denominado por $T \models C \equiv D$.*

Exemplo(Equivalência): Traficante $\equiv$ Pessoa $\sqcap \exists$trafica.Animal. O conceito Traficante define indivíduos que são obrigatoriamente membros da classe Pessoa e caçam pelo menos um Animal.

Definição 8 *Disjunção: Dois conceitos C e D são disjuntos com respeito a T se $C^I \cap D^I = \emptyset$ para todo modelo de I de T.*

Exemplo(Disjunção): AnimalRiscoExtincao $\sqcup$ AnimalSemRiscoExtincao são disjuntos, já que não deve haver animal que esteja ao mesmo tempo em risco e sem risco de extinção.

Definição 9 *Consistência: Uma ABox A é consistente com relação a T, se houver uma interpretação que é um modelo de A e T.*

Essas definições são necessárias para que seja possível inferir conteúdo implícito a partir de declarações explícitas e são fundamentais para o entendimento da lógica como um todo.

2.2 SISTEMAS DE PROVA E O CÁLCULO DE CONEXÕES

Nesta seção são apresentadas algumas definições formais de alguns autores sobre os sistemas de prova, uma apresentação do cálculo de conexões, do cálculo clausal de conexões e do cálculo não-clausal de conexões.

2.2.1 Sistemas de Prova

A teoria da prova é um dos chamados quatro pilares dos fundamentos da matemática. Ela é importante na lógica filosófica, onde os interesses principais estão na ideia de uma semântica, uma ideia que, para ser viável, depende de ideias técnicas. Os sistemas de prova usam um conjunto finito de regras para mostrar a prova completa e estruturada. Definições de alguns autores são apresentadas abaixo.

Definição 10 *Sistema de Prova: Seja L uma linguagem e Λ um conjunto de fórmulas de L. Um sistema de prova S é um par S = (φ, R), onde $\varphi \subseteq \Lambda$ e R é um conjunto finito de regras de inferência (KFOURY; MOLL; ARBIB, 2012).*

Definição 11 *Prova Formal: Uma prova formal de φ a partir de um conjunto de fórmula Γ é uma sequência finita de (α_0, ..., α_n) de fórmulas tal que α_n é φ e para cada $k \leq n$, ou*

(a) α_k está em $\Gamma \cup \Lambda$, onde Λ é um conjunto de axiomas, ou

(b) α_k é obtida por regra de inferência a partir de duas fórmulas anteriores na sequência, isto é, para algum i e k menor que k, α_j é $\alpha_i \rightarrow \alpha_k$ (ENDERTON; ENDERTON, 2001).

Definição 12 *Corretude do Sistema de Prova: Seja Γ um conjunto de fórmulas e φ uma fórmula a ser provada,*

$$se\ \Gamma \vdash \varphi\ então\ \Gamma \models \phi$$

Esse teorema certifica que em todas as interpretações, as fórmulas derivadas são verdadeiras (ENDERTON; ENDERTON, 2001).

Definição 13 *Completude do Sistema de Prova: Seja Γ um conjunto de fórmulas e φ uma fórmula a ser provada,*

$$se\ \Gamma \models \varphi\ então\ \Gamma \vdash \varphi$$

Esse teorema diz que tudo que é semanticamente obtido pode ser também obtido no sistema dedutivo (ENDERTON; ENDERTON, 2001).

Se a prova existe, é dito que φ é dedutível de Γ, ou φ é consequência lógica de Γ, ou que φ é um teorema de Γ, escrito como $\Gamma \vdash \varphi$. O símbolo $\vdash$ é chamado de *turnstile* ou catraca. Abaixo são apresentadas algumas abordagens utilizadas pelos sistemas de prova para provar teoremas.

Prova Direta: Uma prova direta de uma sentença da forma P $\rightarrow$ Q é uma prova que suponhe que P é verdadeiro e então mostra, usando regras de inferência, axiomas previamente aceitos, definições, e equivalências lógicas, que Q é verdadeiro (ROSEN, 1999).

Prova por Contraposição: Para provar P $\rightarrow$ Q considere sua equivalente contrapositiva $\neg$Q $\rightarrow$ $\neg$P verdadeira. Se $\neg$Q $\rightarrow$ $\neg$P é demonstrada verdadeira, então P $\rightarrow$ Q também o é (HAMMACK, 2009).

Prova por Contradição: Para provar P $\rightarrow$ Q, P $\rightarrow$ Q é suposta como falsa, (isto é, suponhe que P é verdadeira e Q é falsa). Se uma contradição é derivada, P $\rightarrow$ Q não pode ser falsa, e portanto deve ser verdadeira (ROSEN, 1999).

2.2.2 Cálculo de Conexões

O Método de Conexões ou Cálculo de Conexões é considerado simples e eficiente, foi introduzido por Bibel (1987). A partir dele, pode-se descrever dois cálculos: o cálculo

clausal de conexões para $\mathcal{ALC}$, que trabalha com fórmulas em Forma Normal Disjuntiva (FND), e o cálculo não-clausal de conexões para $\mathcal{ALC}$, que usa a definição de fórmula com polaridade e matriz não-clausal, ambos os cálculos são apresentados em Palmeira (2017).

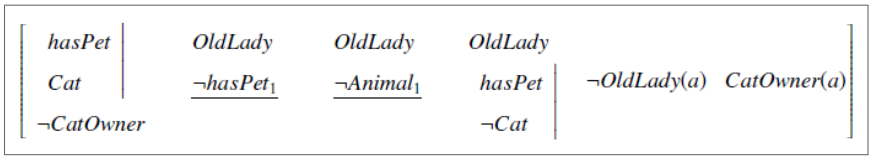
Sendo **KB** uma base de conhecimento e se quer implicar se $\mathbf{KB} \models \alpha$ é válido usando um método direto, por dedução, devemos provar diretamente $\mathbf{KB} \rightarrow \alpha$, em outras palavras, se $\neg\ \mathbf{KB} \vee \{\alpha\}$ for válido. Isso se opõe à refutação clássica, que constrói uma prova testando se $\mathbf{KB} \cup \{\alpha\} \models \bot$. No cálculo de conexões, toda a base de conhecimento be deve ser negada, incluindo predicados instanciados, como $A(a)$, em que a é uma constante ou um indivíduo (FREITAS; VARZINCZAK2, 2018).

2.2.3 Cálculo Clausal de Conexões para $\mathcal{ALC}$

O Cálculo de Conexões trabalha com fórmulas em forma normal disjuntiva (FND), chamado de forma clausal, tendo a forma $C_1 \sqcup ... \sqcup C_n$, onde cada C_i é uma cláusula. As cláusulas são conjunções de literais na forma $L_i \sqcap ... \sqcap L_m$ e cada L_i é um literal.

Dada uma fórmula em forma clausal, ela pode ser escrita como um conjunto de cláusulas, chamada de matriz. A matriz, por sua vez, é uma disjunção de suas cláusulas, que estão dispostas horizontalmente, enquanto os literais são dispostos verticalmente na matriz. A figura 1 mostra a representação gráfica de uma matriz de prova.

Figura 1 – Exemplo de representação gráfica da matriz.



Fonte: (PALMEIRA, 2017)

As próximas definições serão utilizadas para o processo de conversão proposto por Palmeira (2017).

Definição 14 *Literal em $\mathcal{ALC}$: Literais em $\mathcal{ALC}$ são conceitos atômicos ou relações possivelmente negados e/ou instanciados.*

Definição 15 *Disjunção $\mathcal{ALC}$: Uma disjunção em $\mathcal{ALC}$ ou é um literal L, uma disjunção $(E_0 \sqcup E_1)$ ou uma restrição universal $\forall r.E_0$, onde E_0 e E_1 são expressões de conceitos arbitrárias.*

Definição 16 *Conjunção $\mathcal{ALC}$: Uma conjunção $\mathcal{ALC}$ é um literal L, uma conjunção $(E_0 \sqcap E_1)$ ou uma restrição existencial $\exists r.E_0$, onde E_0 e E_1 são expressões de conceitos arbitrárias.*

Definição 17 *Impureza, Disjunção/Conjunção Pura: Impureza em uma fórmula $\mathcal{ALC}$ é uma disjunção em uma conjunção, ou uma conjunção em uma disjunção. Uma disjunção/conjunção pura é uma disjunção/conjunção que não contém impurezas.*

Definição 18 *Forma Normal Disjuntiva (FND) em $\mathcal{ALC}$: Uma fórmula $\mathcal{ALC}$ forma normal disjuntiva ou forma clausal é uma disjunção de conjunções (como $C_1 \sqcup ... \sqcup C_n$), onde cada C_i é uma cláusula. Uma cláusula é uma conjunção pura de literais na forma $L_1 \sqcap ... \sqcap L_m$ e cada L_i é um literal.*

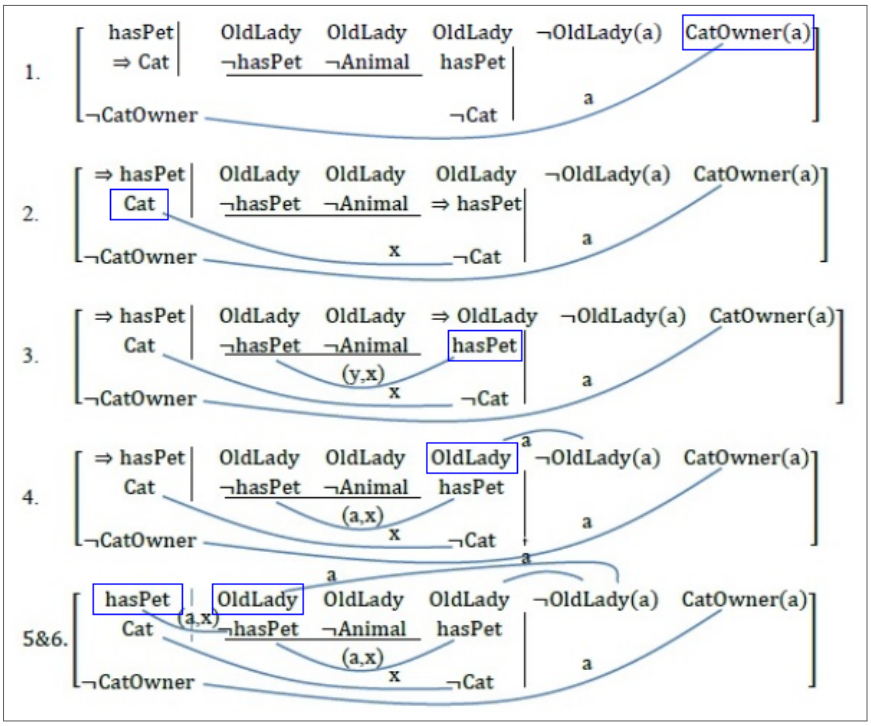
Definição 19 *Forma Normal da Negação (FNN) em $\mathcal{ALC}$ Uma fórmula $\mathcal{ALC}$ está na forma normal da negação se a negação ($\neg$) é aplicada apenas na frente de conceitos (atômicos ou não). Um conceito $\mathcal{ALC}$ arbitrário pode ser transformado em uma FNN equivalente, usando uma combinação das leis de De Morgan e a dualidade entre restrições existenciais e universais $\neg \exists r.C \equiv \forall r.\neg C$ e $\neg \forall r.C \equiv \exists r.\neg C$ (BAADER et al., 2003).*

Definição 20 *Matriz - Uma matriz M é classicamente válida se existir uma multiplicidade μ, um termo de substituição θ e um conjunto de conexões S, de modo que todo caminho através de Mμ contém uma conexão complementar {L1, L2} $\in$ S.*

Definição 21 *Matriz $\mathcal{ALC}$: A matriz $\mathcal{ALC}$ de uma fórmula $\mathcal{ALC}$ em forma normal disjuntiva tem a sua representação como um conjunto C_1,...,C_n, onde cada cláusula C_i tem a forma L_1,...,L_m com literais L_i. Literais envolvidos em uma restrição universal ($\forall r.C$) ou em uma restrição existencial ($\exists r.C$) são sublinhados na matriz da fórmula. Quando uma restrição envolve mais de uma cláusula, seus literais são indexados com o mesmo índice na coluna da matriz.*

A figura 2 mostra um exemplo de prova em representação gráfica da matriz clausal, considerando a fórmula F_1: $\{\exists$hasPet.Cat $\sqsubseteq$ CatOwner, OldLady $\sqsubseteq \exists$hasPet.Animal $\sqcap$ $\forall$hasPet.Cat$\} \models$ OldLady(a) $\sqsubseteq$ CatOwner(a). A matriz está em formato clausal em forma normal disjuntiva.

Figura 2 – Exemplo de Prova Matriz Clausal



Fonte: (FREITAS; OTTEN, 2016)

Definição 22 *Representação Gráfica (Positiva) da Matriz $\mathcal{ALC}$: Em uma representação gráfica da matriz, as cláusulas são colunas, as restrições com índices são representadas por linhas contínuas horizontais, enquanto as restrições sem índices por linhas contínuas verticais. A representação gráfica é dita positiva, quando as cláusulas são representadas como colunas.*

2.2.4 Cálculo Não-Clausal de Conexões para $\mathcal{ALC}$

O Cálculo Não-Clausal de Conexões para $\mathcal{ALC}$ é uma variante do cálculo de θ-Conexões para $\mathcal{ALC}$. Esse cálculo trabalha diretamente na estrutura original da fórmula, sem a necessidade de obtenção da conversão para a Forma Normal Disjuntiva, contendo em sua matriz outras (sub-)matrizes.

O Cálculo Não-clausal de Conexões para $\mathcal{ALC}$ usa algumas definições que são apresentadas a seguir (OTTEN, 2011):

Definição 23 *Fórmula com Polaridade: Uma **fórmula com polaridade**, denotada por F^p, consiste em uma fórmula F e uma polaridade p, em que $p \in \{0, 1\}$, isto é, 0 é positivo e 1 é negativo. Este conceito é utilizado para denotar negação numa matriz, isto é, literais ou matrizes A e $\neg A$ são representados por A^0 e A^1, respectivamente.*

Definição 24 *Matrix Não-Clausal - Uma matriz (não-clausal) é um conjunto de cláusulas, na qual, uma cláusula é um conjunto de literais e matrizes.*

Definição 25 *Matriz $\mathcal{ALC}$ Não-Clausal: Uma matriz $\mathcal{ALC}$ Não-clausal é um conjunto de cláusulas, na qual uma cláusula é um conjunto de literais e matrizes. Seja F uma fórmula $\mathcal{ALC}$ e p uma polaridade. A matriz de F^p, denotada por $M(F^p)$, é definida indutivamente de acordo com a tabela 1, a qual indica como a polaridade é herdada pelas matrizes de uma F^p. A matriz de F é a matriz $M(F^0)$.*

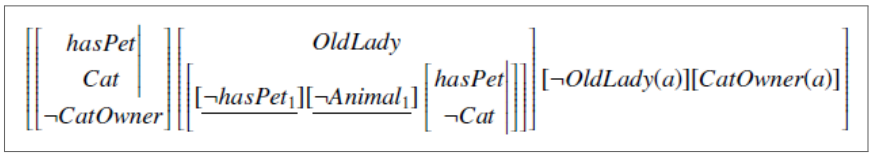
Tabela 1 – Matriz de uma Fórmula $\mathcal{ALC}$ F^p

Tipo	F^p	$M(F^p)$	Tipo	F^p	$M(F^p)$
Atômico	A^0	$\{\{A^0\}\}$	β	$(C \sqcap D)^0$	$\{\{M(C^0), M(D^0)\}\}$
	A^1	$\{\{A^1\}\}$		$(C \sqcup D)^1$	$\{\{M(C^1), M(D^1)\}\}$
α	$(\neg C)^0$	$M(C^1)$		$(C \sqsubseteq D)^1$	$\{\{M(C^0), M(D^1)\}\}$
	$(\neg C)^1$	$M(C^0)$	γ	$(\forall r D)^1$	$\{\{M(\underline{r^0}), M(\underline{D^1})\}\}$
	$(C \sqcap D)^1$	$\{\{M(C^1)\}, \{M(D^1)\}\}$		$(\exists r D)^0$	$\{\{M(\underline{r^0}), M(\underline{D^0})\}\}$
	$(C \sqcup D)^0$	$\{\{M(C^0)\}, \{M(D^0)\}\}$	δ	$(\forall r D)^0$	$\{\{M(\underline{r^1})\}, \{M(\underline{D^0})\}\}$
	$(C \sqsubseteq D)^0$	$\{\{M(C^1)\}, \{M(D^0)\}\}$		$(\exists r D)^1$	$\{\{M(\underline{r^1})\}, \{M(\underline{D^1})\}\}$
	$(C \models D)^0$	$\{\{M(C^1)\}, \{M(D^0)\}\}$			

Fonte: (PALMEIRA, 2017)

Definição 26 *Representação Gráfica Positiva da Matriz: Na representação gráfica (positiva) de uma matriz, suas cláusulas são dispostas horizontalmente, enquanto que os literais e (sub-)matrizes de cada cláusula estão dispostos verticalmente. As restrições são representadas por linhas contínuas; restrições com índices, isto é L_i, j, são representadas com linhas horizontais; restrições sem índices, com linhas verticais. A figura 3 mostra um exemplo de representação gráfica da matriz e a figura 4 mostra um exemplo de representação simplificada da matriz não-clausal*

Figura 3 – Exemplo de Representação Gráfica de Matriz Não-Clausal



Fonte: (PALMEIRA, 2017)

A tabela 2 mostra os passos que são seguidos para se obter a matriz simplificada não-clausal para o exemplo da figura 3.

Tabela 2 – Passos para obter a simplificada matriz não-clausal

F^p	Fórmula/$M(F^p)$
$(G \models H)$	$\left.\begin{array}{c}(\exists(hasPet.Cat) \sqsubseteq CatOwner)\sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))\end{array}\right\} \models (OldLady(a) \sqsubseteq CatOwner(a))$
$(G \models H)^0$	$\left\{\begin{array}{c}\{(((\exists(hasPet.Cat) \sqsubseteq CatOwner)\sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat)))^1\}, \\ \{(OldLady(a) \sqsubseteq CatOwner(a))^0\}\end{array}\right\}$
$(G \sqsubseteq H)^0$	$\left\{\begin{array}{c}\{(((\exists(hasPet.Cat) \sqsubseteq CatOwner)\sqcap \\ (OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat)))^1\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(G \sqcap H)^1$	$\left\{\begin{array}{c}\{\{\{(\exists(hasPet.Cat) \sqsubseteq CatOwner)^1\}, \\ \{(OldLady \sqsubseteq \exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))^1\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(G \sqsubseteq H)^1$	$\left\{\begin{array}{c}\{\{\{\{(\exists(hasPet.Cat))^0,\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ (\exists(hasPet.Animal) \sqcap \forall(hasPet.Cat))^1\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(G \sqcap H)^1$	$\left\{\begin{array}{c}\{\{\{\{(\exists(hasPet.Cat))^0,\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{(\exists(hasPet.Animal))^1\}, \{(\forall(hasPet.Cat))^1\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(\exists GH)^0$	$\left\{\begin{array}{c}\{\{\{\{\{\{\underline{hasPet}^0,\ \underline{Cat}^0\}\},\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{(\exists(hasPet.Animal))^1\}, \{(\forall(hasPet.Cat))^1\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(\forall GH)^1$	$\left\{\begin{array}{c}\{\{\{\{\{\{\underline{hasPet}^0,\ \underline{Cat}^0\}\},\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{(\exists(hasPet.Animal))^1\},\ \{\{\{\underline{hasPet}^0, \underline{Cat}^1\}\}\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$
$(\exists GH)^1$	$\left\{\begin{array}{c}\{\{\{\{\{\{\underline{hasPet}^0,\ \underline{Cat}^0\}\},\ CatOwner^1\}\}\}, \\ \{\{\{OldLady^0,\ \{\{\{\underline{hasPet}^1 1\},\ \{\underline{Animal}^1 1\}\}\}, \{\{\{\underline{hasPet}^0, \underline{Cat}^1\}\}\}\}\}\}\}, \\ \{\{\{OldLady(a)^1\}, \{CatOwner(a)^0\}\}\}\end{array}\right\}$

Fonte: (PALMEIRA, 2017)

Figura 4 – Exemplo de Representação Simplificada de Matriz Não-Clausal

$$\begin{array}{r}\{\ \{\underline{hasPet},\ \underline{Cat},\ \neg CatOwner\}, \\ \{\ OldLady, \{\{\underline{\neg hasPet_1}\}, \{\underline{\neg Animal_1}\},\ \{\underline{hasPet},\ \neg Cat\}\}\ \}, \\ \{\neg OldLady(a)\},\ \{CatOwner(a)\}\ \}\end{array}$$

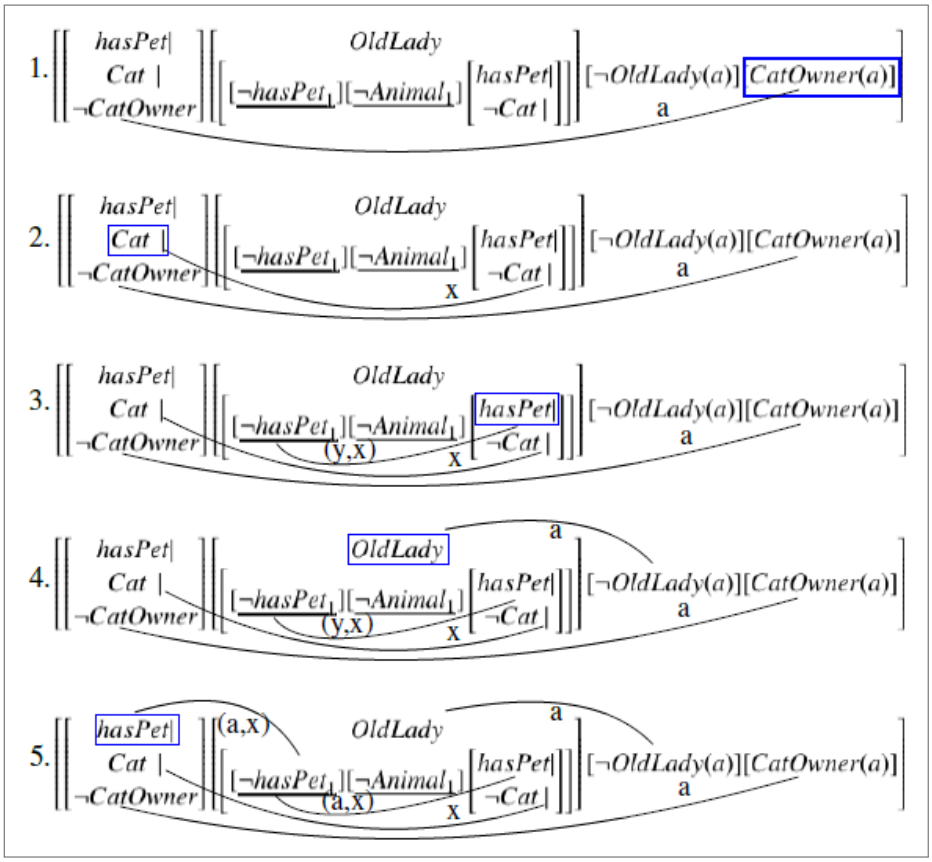
Fonte: (PALMEIRA, 2017)

Definição 27 *Prova de Conexão Não-Clausal - Seja M uma matriz, C seja uma cláusula, e pode ser um conjunto de literais. Uma derivação para C e M com o termo substituição θ no cálculo da conexões não-clausal, em que todas as folhas são axiomas, é chamado de*

prova de conexão (não-clausal) para C e M. Uma prova de conexão (não-clausal) para M é uma prova de conexão não-clausal para ∈, M, ∈.

A definição de fórmula com polaridade, definição 23, é usada sem alteração, mas, a definição de matriz não-clausal é adaptada para contemplar fórmulas $\mathcal{ALC}$ arbitrárias. A figura 5 mostra um exemplo de prova em representação da matriz não-clausal.

Figura 5 – Exemplo de Prova de Matriz Não-Clausal



Fonte: (PALMEIRA, 2017)

2.3 CÁLCULO DE SEQUENTES

2.3.1 Cálculo de Sequentes para Subsunção em $\mathcal{ALC}$

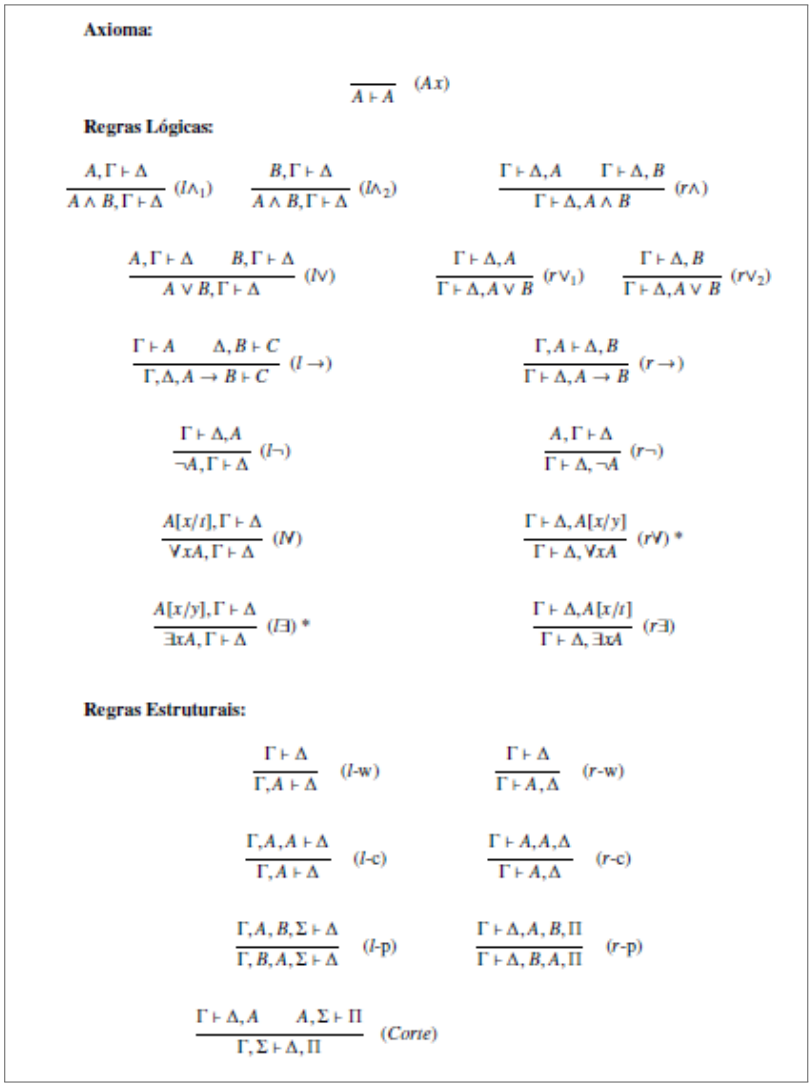
O cálculo de Sequentes foi proposto inicialmente por Gentzen (1935). Um sequente é uma expressão da forma $\Gamma \vdash \Delta$, onde Γ e Δ são sequências finitas de fórmulas, como $\Gamma = \{A_1,...,A\mu\}$ e $\Delta = \{B_1,...,B\nu\}$, ambas as expressões podem ser vazias. Γ é o antecedente e Δ o sucedente do sequente. O sequente $\{A_1,...,A\mu\} \vdash \{B_1,...,B\nu\}$ é considerado verdadeiro se $A_1 \sqcap ... \sqcap A\mu$ implica em $B_1 \sqcup ... \sqcup B\nu$.

De acordo com Borgida, Franconi e Horrocks (2000), o cálculo de sequentes axiomatiza a relação de consequência lógica (entailment). Com isso, pode-se observar um paralelo com a relação de subsunção. É proposto por Borgida, Franconi e Horrocks (2000) um

cálculo de sequentes para inferências de subsunção em $\mathcal{ALC}$. Nesse cálculo não existem as regras da implicação, dada a relação com a subsunção mencionada acima, os termos não são movidos de um lado para o outro do torniquete durante a prova, ao contrário do Cálculo de Sequentes para Lógica de Primeira Ordem (LPO), preservando assim a estrutura da subsunção original. Para isso, regras adicionais foram criadas nas quais a negação é inserida na frente de cada construtor e assim eliminam-se as regras de negação (l¬, r¬), as quais exigem mudança dos antecedentes do sequente para sucedentes e vice-versa.

No Cálculo de Sequentes, as provas ou deduções são rotuladas como árvores finitas com uma única raiz, com axiomas nos nós superiores e cada rótulo do nó conectado com os rótulos dos nós sucessores de acordo com uma das regras. A figura 6 mostra as regras propostas por Gentzen (1935) para LPO.

Figura 6 – Regras para Cálculo de Sequentes em LPO.

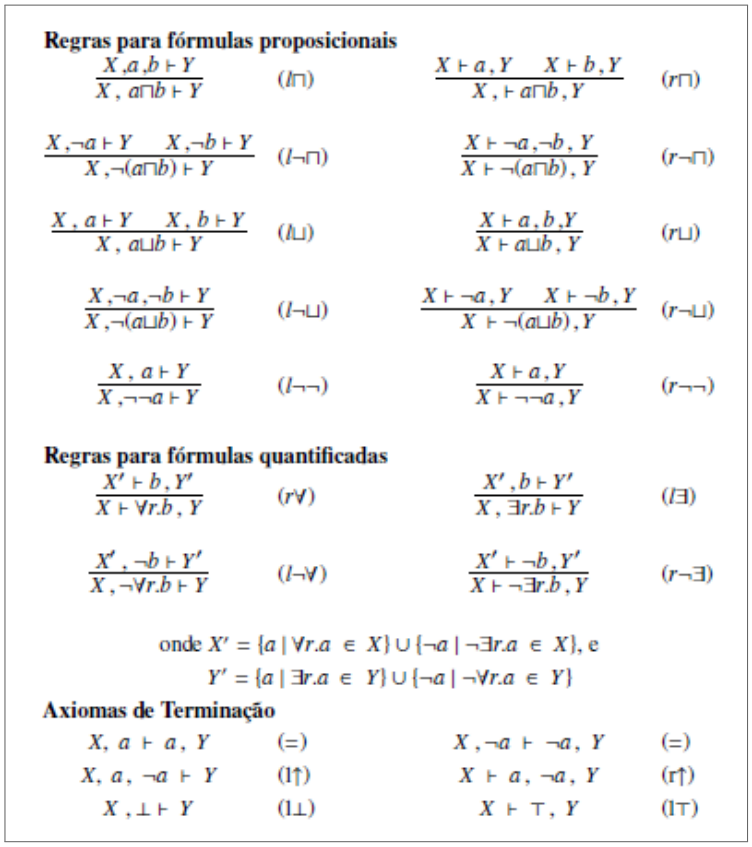


Fonte: (GENTZEN, 1935)

Em Borgida, Franconi e Horrocks (2000), foram propostas regras organizadas em três

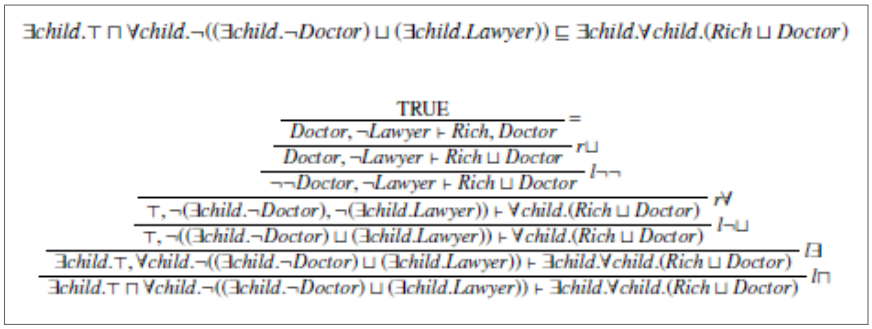
partes para o cálculo de sequentes para $\mathcal{ALC}$: Regras para fórmulas proposicionais, regras para fórmulas quantificadas e axiomas de terminação. As duas primeiras descrevem conjuntos de regras, enquanto a última descreve um conjunto de axiomas ($\wedge$ e $\vee$ são substituídos por seus correspondentes $\sqcap$ e $\sqcup$ em $\mathcal{ALC}$. A figura 7 mostra as regras e os axiomas para $\mathcal{ALC}$ e, em seguida, é apresentado um exemplo de prova, figura 8.

Figura 7 – Regras para Cálculo de Sequentes para Subsunção $\mathcal{ALC}$.



Fonte: (BORGIDA; FRANCONI; HORROCKS, 2000)

Figura 8 – Exemplo e Prova em $\mathcal{ALC}$.



Fonte: (GENTZEN, 1935)

Regras para fórmulas proposicionais: as regras $\sqcap$ e $\sqcup$ são duplicadas pela adição das regras de negação ($\neg\sqcap$; $\neg\sqcup$), enquanto as regras $\neg$ são modificadas para incluir mais uma

negação ($\neg\ \neg$), se tornando duplamente negadas;

Regras para fórmulas quantificadas: as regras com quantificadores também possuem a negação na frente do $\forall$ e $\exists$, gerando as regras l$\neg\forall$ e r$\neg\exists$. Além disso, uma condição é explicitamente considerada para a aplicação das regras $\forall$ e $\exists$. A condição declara que a regra é aplicável se todas as fórmulas universais e existenciais homólogas ($\forall h.C$ e $\exists h.C$ são homólogas, $\forall h.C$ e $\exists f.C$)não estão reunidas juntas nos lados esquerdo e direito do sequente na pré-condição; a regra é então aplicada uma única vez;

Axiomas de terminação: neste cálculo existem seis possíveis axiomas de terminação, ao contrário do cálculo de sequentes padrão no qual todos os axiomas de terminação podem ser reduzidos a X; a $\vdash$ a; Y pela aplicação das regras $\neg$. A aplicação das regras $\neg$ força a mudança das fórmulas do antecedente do sequente para o sucedente ou vice e versa até chegar a X; a $\vdash$ a; Y, procedimento que é evitado nesse cálculo. Portanto, os axiomas adicionais de terminação são necessários para garantir que as fórmulas nunca sejam deslocadas de um lado para o outro do sequente.

Embora não seja declarado explicitamente, o cálculo contém uma regra de corte, e o teorema de eliminação de corte é válido neste caso é indicado abaixo.

Teorema da Eliminação do Corte (GIRARD; TAYLOR; LAFONT, 1989): Seja S um conjunto de sequentes (axiomas) e s um sequente individual. $S \vdash_{SC} s$, se e somente se, existe uma prova em SC de s cujas folhas são axiomas lógicos ou sequentes obtidos pela substituição de seqüências de sequentes pentencentes a S, onde a regra do corte, $\frac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$, é somente aplicada com uma premissa sendo um axioma.

Essa proposta utiliza o cálculo de sequentes para $\mathcal{ALC}$. Sua principal vantagem é preservar a estrutura original dos dois conceitos e que os conceitos nunca são transferidos de antecedente para sucedente ou vice-versa. Esse cálculo considera o uso da subsunção, considerada a inferência chave para lógica de descrições (BORGIDA; FRANCONI; HORROCKS, 2000) e, é equivalente a implicação em LPO.

3 CONVERSÃO DE PROVAS $\mathcal{ALC}$ EM CONEXÕES PARA SEQUENTES

Este capítulo mostra o método de conversão de provas $\mathcal{ALC}$ construídas com o método de conexões não-clausal em provas em sequentes $\mathcal{ALC}$, proposto por Palmeira (2017). Esse método é baseado no método de prova para Lógica de Primeira Ordem proposto por Otten e Kreitz (1995), ele difere por utilizar uma notação sem variável e construtores específicos da Lógica de Descrições e pela adição de uma substituição para lidar com instâncias e trabalhar com regras próprias do cálculo de sequentes para $\mathcal{ALC}$.

Nesse método foram utilizadas as noções de caminhos e conexões propostas por Otten e Kreitz (1995). Serão apresentadas definições sobre os conceitos utilizados, conforme Palmeira (2017).

3.1 DEFINIÇÕES PRELIMINARES

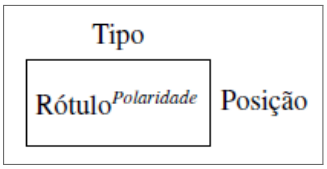
Definição 28 *Árvore de Fórmula, Posição, Rótulo, Polaridade, Tipo: Uma árvore de fórmula é uma representação de uma fórmula F como árvore, onde cada nó poderá ter até dois nós filhos. Cada nó possui os seguintes dados:*

- *Posição: uma posição é um índice na forma a_0, a_1, a_2, a_3...;*
- *Rótulo: um rótulo ou é um conectivo($\sqcap$,$\sqcup$,$\neg$,$\rightarrow$)ou um quantificador ou de um predicado, se é uma (sub-)fórmula atômica. Nós rotulados com predicados são folhas da árvore, enquanto os demais nós são chamados de nós internos;*
- *Polaridade: a polaridade (0 ou 1) de um nó é determinada pelo rótulo e polaridade de seu nó pai. O nó raiz, primeiro nó da árvore, tem polaridade 0;*
- *Tipo: o tipo de um nó é representado por uma das letras gregas: α,β,α',β', γ e δ. Ele é determinado por seu rótulo e sua polaridade. Nós folha não têm tipo. A polaridade e o tipo de um nó são definidos na figura 11. Por exemplo, na primeira linha dessa tabela, (A $\sqcap$ B) 1 significa que o nó rotulado com $\sqcap$ e polaridade 1 tem tipo α e seus nós sucessores têm polaridade 1.*

A figura 9 mostra a representação de um nó interno e a figura 10 mostra a representação de um nó folha.

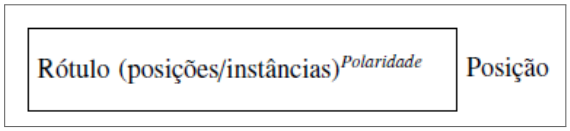
Definição 29 *Caminho entre dois nós: Sejam dois nós, um com posição a_i e o outro com posição a_j. Um caminho entre esses dois nós através de uma árvore de fórmula F, é um subconjunto de posições dos nós de sua árvore que ligam esses dois nós na forma a_i,a_2,...,a_{j-1}, a_j.*

Figura 9 – Representação de um nó interno



Fonte: (PALMEIRA, 2017)

Figura 10 – Representação de um nó folha



Fonte: (PALMEIRA, 2017)

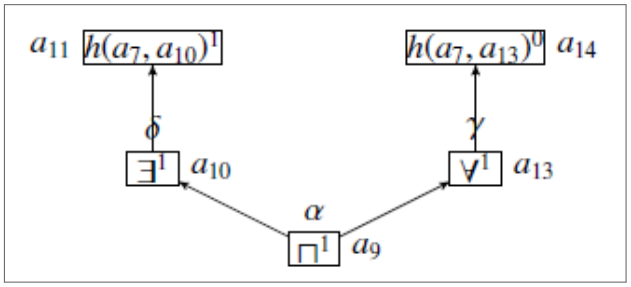
Figura 11 – Polaridade e tipos dos nós para $\mathcal{ALC}$

Tipo α			Tipo β			Tipo δ		
$(A \sqcap B)^1$	A^1	B^1	$(A \sqcap B)^0$	A^0	B^0	$(\forall r A)^0$	r^1	A^0
$(A \sqcup B)^0$	A^0	B^0	$(A \sqcup B)^1$	A^1	B^1	$(\exists r A)^1$	r^1	A^1
$(\neg A)^1$	A^0							
$(\neg A)^0$	A^1							
Tipo α'			Tipo β'			Tipo γ		
$(A \sqsubseteq B)^0$	A^1	B^0	$(A \sqsubseteq B)^1$	A^0	B^1	$(\forall r A)^1$	r^0	A^1
$(A \models B)^0$	A^1	B^0				$(\exists r A)^0$	r^0	A^0

Fonte: (PALMEIRA, 2017)

A figura 12 mostra um exemplo de caminho entre um nó de posição a_{11} com rótulo h^1 até o nó a_{14} com rótulo h^0. Percorrendo a árvore o caminho é $\{a_{11}, a_{10}, a_9, a_{13}, a_{14}\}$.

Figura 12 – Exemplo de Caminho



Fonte: (PALMEIRA, 2017)

Definição 30 *Ordem de redução $\lhd$: O fecho transitivo da união de $\sqsubseteq_\delta$, $\sqsubseteq_{\beta'}$ e $\prec$ é chamado de ordem de redução $\lhd$, isto é, $\lhd = (\prec \cup \sqsubseteq_\delta \cup \sqsubseteq_{\beta'})^+$. A expressão $v \lhd u$ significa que o nó rotulado por v deve ser reduzido antes do nó rotulado por u na prova em sequentes.* $\lhd$

determina a ordem de redução dos nós da árvore, bem como ajuda a determinar quais e em que ordem as regras do cálculo de sequentes devem ser aplicadas.

São definidas três substituições de posições, para substituir posições associadas aos rótulos folha:

- substituição σ_δ: substitui posições do tipo γ por posições do tipo δ;
- substituição $\sigma_{\beta'}$: substitui posições do tipo β', γ, δ por instâncias ou posições do tipo β';
- substituição combinada σ_{Final}: consiste de uma combinação das duas primeiras.

Definição 31 *Substituição de posições σ_δ, relação de ordenamento $\subset_\delta$: Uma substituição de posições σ_δ é um mapeamento do conjunto Γ de posições dos nós do tipo γ para o conjunto Δ de posições de nós do tipo δ. A substituição σ_δ induz uma relação de ordenamento parcial $\subset_\delta$ em $\Delta \times \Gamma$ da seguinte forma: seja $u \in \Gamma$ e $v \in \Delta$; se $\sigma_\delta(u) = p$ então $v \subset_\delta u$ para todo $v \in \Delta$ ocorrendo na posição p.*

Como as regras de sequentes r$\forall$ e l$\exists$ e suas homólogas l$\neg\forall$ e r$\neg\exists$ são restritas a condição de autovariável, a relação v $_\delta$ u expressa que o nó rotulado por v deve ser reduzido antes de reduzir um rotulado por u.

Definição 32 *Substituição de posições $\sigma_{\beta'}$: As posições dos nós do tipo β', γ e δ, assim como instâncias aparecem em fórmulas atômicas. Consequentemente, uma substituição de posições $\sigma_{\beta'}$ é um mapeamento do conjunto $B'/\Gamma/\Delta$ de posições dos nós do tipo $\beta'/\gamma/\delta$ para instâncias ou posições dos nós do tipo $\beta_\prime$. Seja u um nó folha com posições dos nós do tipo $\beta'/\gamma/\delta$ associadas a seu rótulo e v $\exists$ B' ; se $\sigma_{\beta'}(u) = p$, onde p $\exists B'$ ou p é uma instância.*

Definição 33 *Substituição σ_{Final}: Uma substituição σ_{Final} consiste de uma substituição σ_δ e uma substituição $\sigma_{\beta'}$, onde $\sigma_{Final} := \sigma_\delta \cup \sigma_{\beta'}$*

Definição 34 *Conexão, conexão σ_{Final}-complementar : Uma conexão é um par de nós folha rotulados com o mesmo símbolo de predicado e a mesma posição associada ao rótulo ou mesma instância, mas com diferentes polaridades. Se eles são idênticos sob σ_{Final}, a conexão é chamada de conexão σ_{Final}-complementar.*

Definição 35 *Substituição σ_{Final} admissível: Uma substituição σ_{Final} é admissível se a ordem de redução $\lhd$ o é reflexiva. Neste caso é possível construir uma prova no cálculo de sequentes.*

Uma correspondência entre rótulo, polaridade e tipo de um nó com as regras do sequentes é mostrada na tabela 3. Tal correspondência é útil para a construção da prova em sequentes, onde a polaridade auxilia na identificação da regra. A polaridade 1 representa uma regra à esquerda (left ou l), enquanto a polaridade 0, à direita (right ou r), para os casos em que há uma regra associada. Por exemplo, na primeira linha da tabela 3, para o nó $\sqcup 1$ a regra é $l\sqcup$, e para o nó $\sqcap 0$ a regra é $r\sqcap$. Para os casos em que nós internos são precedidos por um nó rotulado por uma negação, a correspondência é dada pela tabela 4.

Tabela 3 – Correspondência entre rótulo, polaridade e tipo de um nó, não precedido por um nó rotulado por uma negação, com as regras do sequentes

Tipo α	Regra
$\sqcap^1$	$1\sqcap$
$\sqcup^0$	$r\sqcup$
$\neg^1$	$\varnothing$
$\neg^0$	$\varnothing$

Tipo β	Regra
$\sqcap^0$	$r\sqcup$
$\sqcup^1$	$1\sqcap$

Tipo δ	Regra
$\forall^0$	$r\forall$
$\exists^1$	$1\exists$

Tipo α'	Regra
$\sqsubseteq^0$	$\varnothing$
$\models^0$	$\varnothing$

Tipo β'	Regra
$\sqsubseteq^1$	$\frac{\Gamma \vdash \Delta, A \qquad A, \Sigma \vdash \Pi}{\Gamma, \Sigma \vdash \Delta, \Pi}$

Tipo γ	
$\forall^1$	$\varnothing$
$\exists^0$	$\varnothing$

Fonte: (PALMEIRA, 2017)

Tabela 4 – Correspondência entre rótulo, polaridade e tipo de um nó, precedido por um nó rotulado por uma negação, com as regras do sequentes

Tipo α	Regra
$\neg^1$	$r\neg\neg$
$\neg^0$	$1\neg\neg$
$\sqcap^1$	$r\neg\sqcap$
$\sqcup^0$	$1\neg\sqcup$

Tipo β	Regra
$\sqcap^0$	$1\neg\sqcap$
$\sqcap^1$	$r\neg\sqcup$

Tipo δ	Regra
$\forall^0$	$1\neg\forall$
$\exists^1$	$r\neg\exists$

Fonte: (PALMEIRA, 2017)

3.2 PROCESSO DE CONVERSÃO

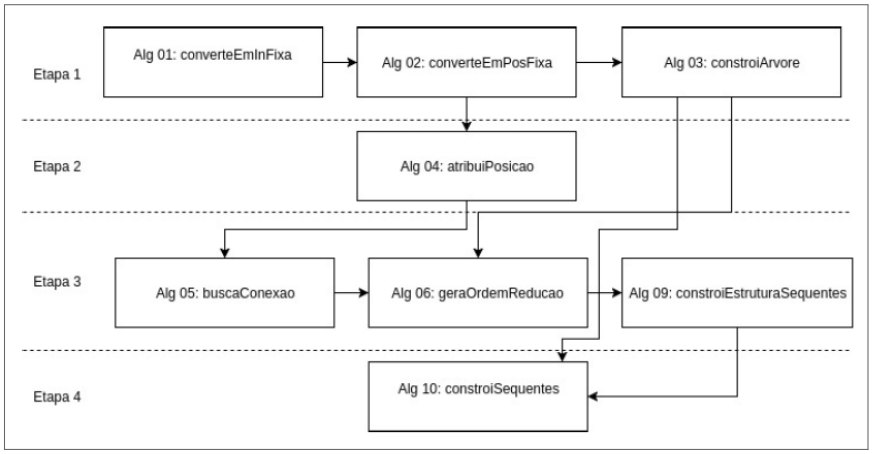
Esta seção apresenta o método de conversão de provas $\mathcal{ALC}$ construídas com o método de conexões não-clausal, seção 2.2.4, em provas no cálculo de sequentes para $\mathcal{ALC}$, apresentado na seção 2.3.1. O método disponibiliza a junção do poder de expressividade da Lógica de Descrições com o bom desempenho de provadores de raciocínio automático. Esse método apresenta uma compreensão mais amigável, auxiliando de forma mais facilitada a interação com usuários em geral (PALMEIRA, 2017).

Dada uma fórmula $\mathcal{ALC}$ e a sua prova em conexões na matriz não-clausal, gerada pelo método de conexões não-clausal, mostrado aqui na seção 2.2.4, o processo de conversão

transforma a prova em conexões para uma prova em sequentes, utilizando o cálculo de sequentes para $\mathcal{ALC}$. São propostas por Palmeira (2017) quatro etapas para o processo de conversão, conforme é apresentado no diagrama da figura 13, são elas:

- Etapa 1- Construção da árvore de fórmula: Nessa etapa, uma representação sintática em forma de árvore é construída para a fórmula de entrada;
- Etapa 2- Atribuição de posições aos elementos da matriz: Como os elementos da matriz de prova correspondem aos predicados existentes na fórmula, que por sua vez também correspondem aos nós folha na árvore de fórmula, essa etapa atribui a cada elemento da matriz a posição do predicado correspondente;
- Etapa 3- Construção da estrutura da prova (parcial) em sequentes: Para cada conexão da matriz, a árvore de fórmula é examinada em busca dos nós folha que correspondem à conexão. Os caminhos entre o nó raiz e esses nós na árvore são analisados, a fim de determinar a ordem dos nós a ser trabalhada e com isso construir uma estrutura da prova (parcial) em sequentes. Essa estrutura fornece informações sobre a ordem de redução C, que ajuda a determinar as regras que devem ser aplicadas, e sobre a existência de ramificação da prova, dada pela identificação de nós do tipo β e β'.
- Etapa 4- Construção da prova completa em sequentes: Esta é a quarta e última etapa do processo onde uma prova completa em sequentes é construída a partir da estrutura da prova (parcial) em sequentes e da correspondência entre o nó e as regras do sequentes, tabela 3 e tabela 4.

Figura 13 – Etapas Processo de Conversão de Provas para Sequentes $\mathcal{ALC}$

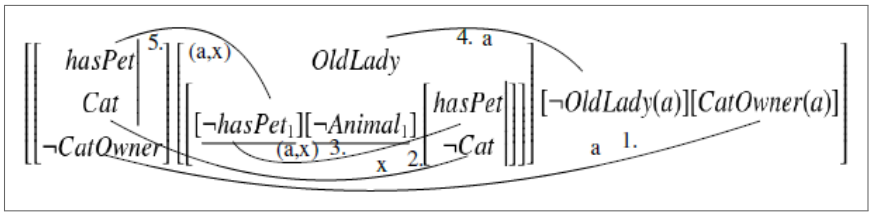


Fonte: (PALMEIRA, 2017)

Considere o exemplo de fórmula F_1 para conversão no método proposto por Palmeira (2017) e a representação gráfica da prova não-clausal para F_1 a figura 14.

Exemplo de Fórmula F_1: {∃hasPet.Cat ⊑ CatOwner, OldLady ⊑ ∃hasPet.Animal ⊓ ∀hasPet.Cat} ⊨ OldLady(a) ⊑ CatOwner(a).

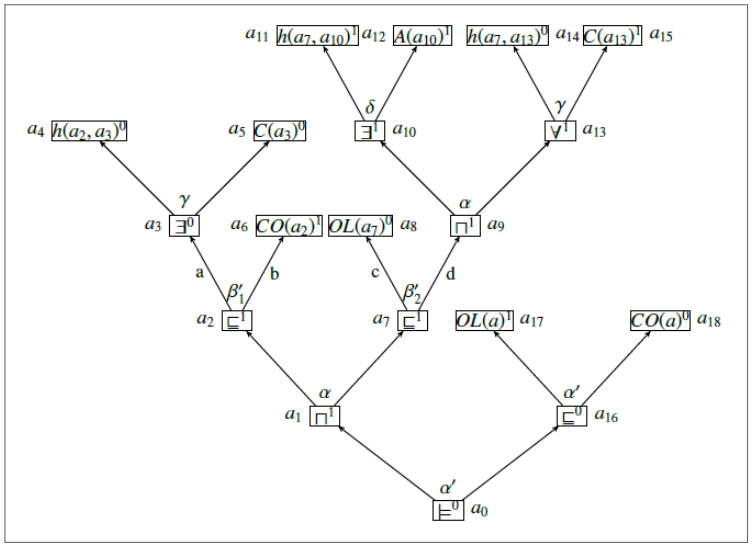
Figura 14 – Representação Gráfica da Prova Não-Clausal para F1



Fonte: (PALMEIRA, 2017)

Etapa 1: Nessa etapa é construída a árvore de fórmula de F_1 contendo todas as informações necessárias, conforme mostra a figura 15.

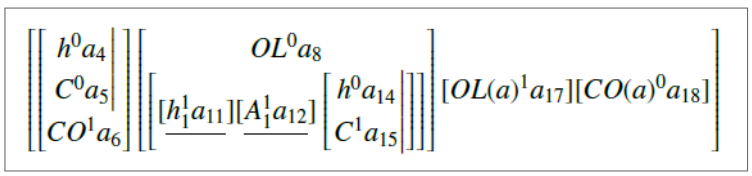
Figura 15 – Árvore de Fórmula para F_1



Fonte: (PALMEIRA, 2017)

Etapa 2: Nessa etapa é obtida a matriz com os elementos contendo a posição nó correspondente na árvore de fórmula, conforme mostra a figura 16.

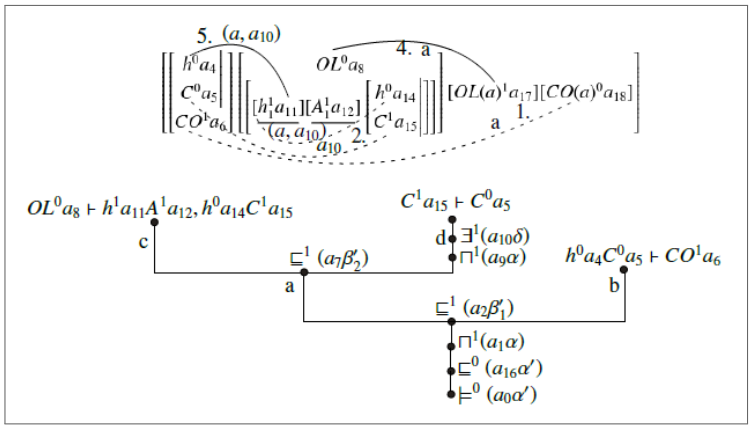
Figura 16 – Representação Gráfica da Matriz Não-Clausal para F_1 com Posições.



Fonte: (PALMEIRA, 2017)

Etapa 3: Nessa etapa as conexões são analisadas e é gerada a estrutura parcial em sequentes, conforme mostra a figura 17.

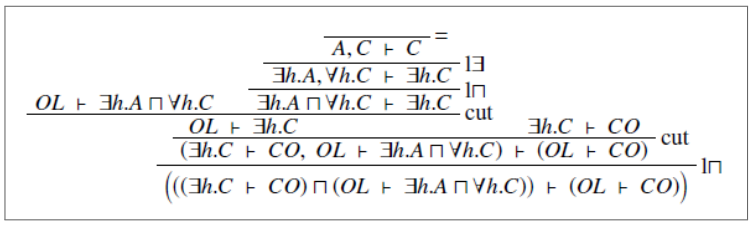
Figura 17 – Matriz e Conexões e Estrutura Parcial em Sequentes.



Fonte: (PALMEIRA, 2017)

Etapa 4: Nessa etapa, são omitidas as intâncias obtidas através do método não-clausal. A estrutura parcial obtida na etapa 3 é percorrida e a cada nó é encontrada a regra correspondente a ser aplicada. A figura 18 mostra a saída da construção completa da prova em sequentes para F_1.

Figura 18 – Prova Final em Sequentes.



Fonte: (PALMEIRA, 2017)

4 IMPLEMENTAÇÃO DE CONVERSÃO DE PROVAS PARA O CÁLCULO DE SEQUENTES

Esse capítulo explica as etapas do processo de conversão implementado nesse trabalho. É exemplificado na seção 4.2 o processamento de uma fórmula utilizando os algoritmos criados em JAVA. Palmeira (2017) calculou a complexidade dos algoritmos que foram implementados nesse trabalho (ANEXO B) e, no pior dos casos, ela é exponencial.

4.1 PROCESSO DE CONVERSÃO

O processo de conversão implementado neste trabalho é formalizado por Palmeira (2017). Após a entrada da fórmula para conversão, as etapas aplicadas são:

- Etapa 1: Nesta etapa a fórmula a ser convertida é obtida e convertida para a forma infixa. Posteriormente, a forma infixa é convertida para a forma posfixa e, em seguida, é gerada uma representação sintática em forma de árvore para a fórmula de entrada. Para a construção da árvore de prova, é preciso consultar a tabela 5 para identificar o tipo de cada nó;
- Etapa 2: É obtida como entrada a matriz de prova com os elementos correspondentes aos predicados existentes na fórmula de entrada, que são os nós folha da árvore gerada na etapa 1. Nessa etapa são atribuídas as posições do predicado para cada elemento da matriz.

Tabela 5 – Tipos e Polaridades

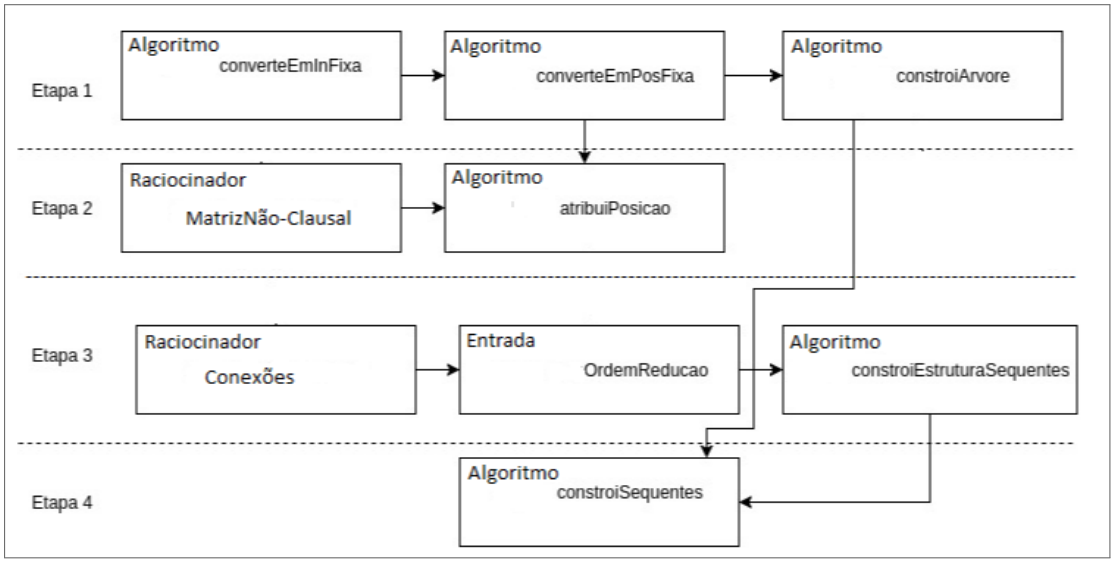
Tipo	**Con**	**Pol**	**PolNoE**	**PolNoD**
α	$\sqcap$	1	1	1
α	$\sqcup$	0	0	0
α	$\neg$	1		0
α	$\neg$	0		1
α'	$\sqsubseteq$	0	1	0
α'	$\models$	0	1	0
β'	$\sqsubseteq$	1	0	1
β	$\sqcap$	0	0	0
β	$\sqcup$	1	1	1
δ	$\forall$	0	1	0
δ	$\exists$	1	1	1
γ	$\forall$	1	0	1
γ	$\exists$	0	0	0

Fonte: (PALMEIRA, 2017)

- Etapa 3: São obtidas, através de um raciocinador, as conexões da matriz. Em seguida, é obtida a entrada da ordem de redução dos nós para gerar uma estrutura parcial da prova em sequentes.
- Etapa 4: Nesta etapa é construída a prova completa em sequentes a partir da estrutura parcial da prova gerada na etapa 3 e da correspondência entre o nó e a devida regra de sequente.

A figura 19 mostra um diagrama das etapas do processo de conversão implementado nesse trabalho.

Figura 19 – Etapas do Processo de Conversão Implementado.



Fonte: O Autor (2019)

Após a análise de alguns testes, foi verificado que o pseudocódigo dos algoritmos "constroiSequentes"e "aplicaRegraCorte"(ANEXO B), precisavam ser modificados para que fosse possível gerar a prova no cálculo de sequentes. A implementação desses e dos demais algoritmos implementados pode ser vista no APENDICE A.

4.2 PROCESSAMENTO DA CONVERSÃO

Nesta seção é apresentado um exemplo de processamento da conversão implementada.

Exemplo1: Considere a fórmula F_1 como exemplo de entrada:

$\{\exists hasPet.Cat \sqsubseteq CatOwner, OldLady \sqsubseteq \exists hasPet.Animal \sqcap \forall hasPet.Cat\} \models OldLady(a) \sqsubseteq CatOwner(a)$

Para o processo de conversão, são necessárias algumas entradas externas ao sistema de conversão apresentado: A matriz de prova não-clausal, as conexões e a ordem de redução do sequente.

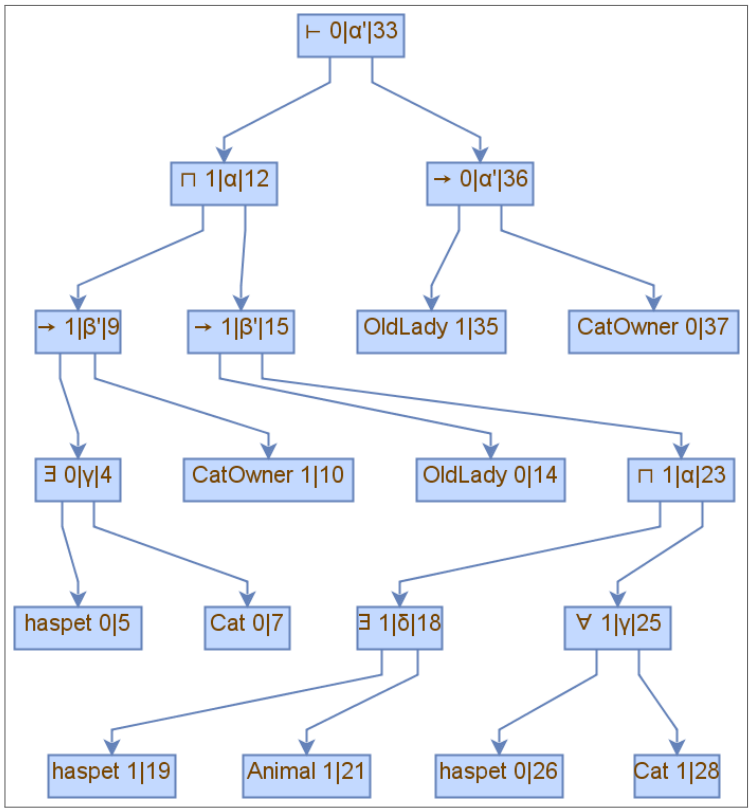
Etapa1: A fórmula de entrada é dividada em tokens e o índice de cada token é numerado. Em seguida, a fórmula é convertida para a forma infixa e, posteriormente, para a forma pós-fixa. Por fim, é gerada a árvore de prova da fórmula, figura 20. Como a implicação tem sua correspondência lógica com a subsunção, para facilitar a geração dos algoritmos, o símbolo de subsunção foi substituído pelo símbolo da implicação.Os nós internos estão representados na seguinte ordem: rótulo polaridade|tipo|posição. Os nós folha estão representados na ordem: rótulo polaridade|posição.

Fórmula de Entrada com Índices:
(|0 (|1 (|2 (|3 ∃|4 haspet|5 .|6 Cat|7)|8 ⊑|9 CatOwner|10)|11 ⊓|12 (|13 OldLady|14 ⊑|15 (|16 (|17 ∃|18 haspet|19 .|20 Animal|21)|22 ⊓|23 (|24 ∀|25 haspet|26 .|27 Cat|28)|29)|30)|31)|32 ⊨|33 (|34 OldLady|35 ⊑|36 CatOwner|37)|38)|39

Fórmula Infixa:
(|0 (|1 (|2 (|3 haspet|5 ∃|4 Cat|7)|8 ⊑|9 CatOwner|10)|11 ⊓|12 (|13 OldLady|14 ⊑|15 (|16 (|17 haspet|19 ∃|18 Animal|21)|22 ⊓|23 (|24 haspet|26 ∀|25 Cat|28)|29)|30)|31)|32 ⊨|33 (|34 OldLady|35 ⊑|36 CatOwner|37)|38)|39

Fórmula Pós-fixa:
haspet|5 Cat|7 ∃|4 CatOwner|10 ⊑|9 OldLady|14 haspet|19 Animal|21 ∃|18 haspet|26 Cat|28 ∀|25 ⊓|23 ⊑|15 ⊓|12 OldLady|35 CatOwner|37 ⊑|36 ⊨|33

Figura 20 – Árvore de Prova de F_1



Fonte: O Autor (2019)

Etapa2: É obtida como entrada a matriz de prova e, em seguida, são atribuídas as posições dos elementos da matriz de prova:

Matriz de Prova:
{{1,haspet,0,null,5;2,Cat,0,null,2;3,CatOwner,1,null,1},{4,OldLady,0,null,4;
{5,haspet,1,null,{3,5}},{6,Animal,1,null,null}{7,haspet,0,null,3;8,Cat,1,null,2}},
{9,OldLady,1,null,4},{10,CatOwner,0,null,1}}

Matriz de Prova com Posições:
{{1,haspet,0,4,5;2,Cat,0,7,2;3,CatOwner,1,10,1},{4,OldLady,0,14,4;{5,haspet,1,19,{3,5}},
{6,Animal,1,21,null}{7,haspet,0,26,3;8,Cat,1,28,2}},{9,OldLady,1,35,4},
{10,CatOwner,0,37,1}}

Etapa3: São obtidas as conexões, tabela 6, e a ordem de redução do sequente. Em seguida, é gerada a estrutura da prova parcial em sequentes, figura 21. Os nós internos estão representados na seguinte ordem: rótulo polaridade|posição|tipo. Os nós folha estão representados na ordem: rótulo polaridade|posição. A vírgula separa os nós folha com mais de um rótulo.

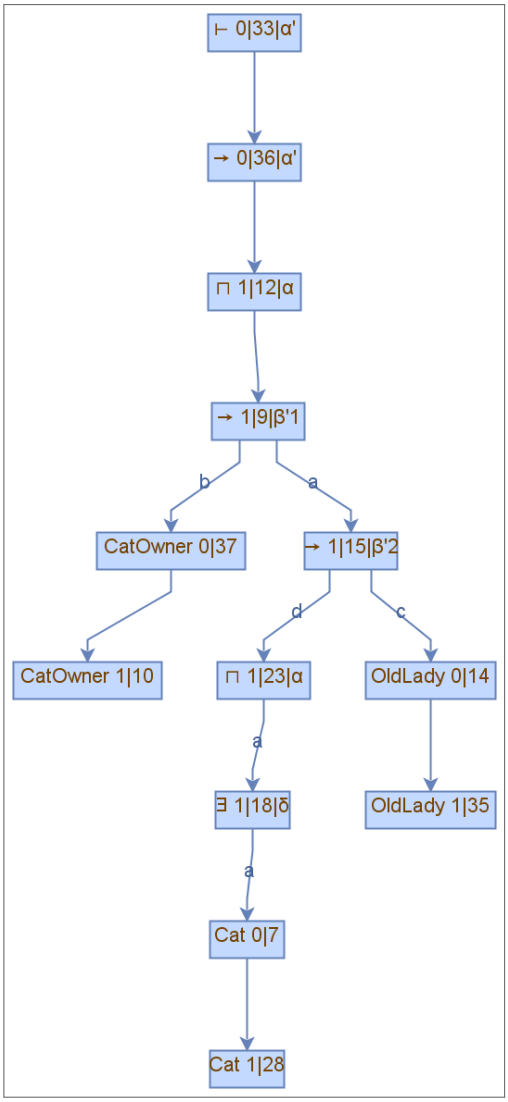
OrdemReducao = {33,36,12,9,{37,10},15,23,18,{7,28},{14, 35}};

Tabela 6 – Conexões de Entrada

Conexão	Posição 1	Posição 2
1	37	10
2	7	28
3	26	19
4	14	35
5	5	19

Fonte: O Autor (2019)

Figura 21 – Estrutura Parcial da Prova em Sequentes



Fonte: O Autor (2019)

Etapa4: Nesta etapa é construída a prova final em sequentes. De acordo com as tabelas 3 e 4, não existem regras associadas para os dois primeiros nós da estrutura parcial

da prova, então, a primeira regra aplicada é a l⊓:

(02) ∃haspet.Cat → CatOwner, OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat ⊢ (OldLady → CatOwner)
(01) ((∃haspet.Cat → CatOwner) ⊓ (OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat)) ⊢ (OldLady → CatOwner)

Em seguida, o próximo nó é do tipo β' e é reduzido com a aplicação da regra do corte sobre (OldLady ⊢ CatOwner), logo, a prova é dividida nos ramos 'a' e 'b'. O ramo 'b' é fechado em seguida com (∃haspet.Cat → CatOwner), devendo seguir com o ramo 'a'. A separação do corte é dvividida por ||.

(03) OldLady ⊢ ∃haspet.Cat||∃haspet.Cat ⊢ CatOwner
(02) ∃haspet.Cat → CatOwner, OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat
⊢ (OldLady → CatOwner)
(01) ((∃haspet.Cat → CatOwner) ⊓ (OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat))
⊢ (OldLady → CatOwner)

O próximo nó, seguindo pelo ramo 'a', é do tipo β' e é reduzido com a regra do corte mais uma vez, gerando os ramos 'c' e 'd'. A regra é aplicada sobre (OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat). O ramo 'c' é fechado e segue a construção pelo ramo 'd':

(04) OldLady ⊢ ∃haspet.Animal ⊓ ∀haspet.Cat||∃haspet.Animal ⊓ ∀haspet.Cat ⊢ ∃haspet.Cat
(03) OldLady ⊢ ∃haspet.Cat||∃haspet.Cat ⊢ CatOwner
(02) ∃haspet.Cat → CatOwner, OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat ⊢ (OldLady → CatOwner)
(01) ((∃haspet.Cat → CatOwner) ⊓ (OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat)) ⊢ (OldLady → CatOwner)

No ramo 'd' é aplicada a regra l⊓ e em seguida a regra l∃, fechando assim o ramo 'd'. Com a aplicação das regras, é concluída a prova em sequentes:

(06) Animal, Cat ⊢ Cat
(05) ∃haspet.Animal, ∀haspet.Cat ⊢ ∃haspet.Cat
(04) OldLady ⊢ ∃haspet.Animal ⊓ ∀haspet.Cat||∃haspet.Animal ⊓ ∀haspet.Cat ⊢ ∃haspet.Cat
(03) OldLady ⊢ ∃haspet.Cat||∃haspet.Cat ⊢ CatOwner
(02) ∃haspet.Cat → CatOwner, OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat ⊢ (OldLady → CatOwner)
(01) ((∃haspet.Cat → CatOwner) ⊓ (OldLady → ∃haspet.Animal ⊓ ∀haspet.Cat)) ⊢ (Ol-

dLady → CatOwner)

Exemplo2: Considere a fórmula F_2 como exemplo de entrada:

{(Socrates ⊑ Homem) ⊓ (Homem ⊑ Masculino)} ⊨ (Socrates ⊑ Masculino)

Etapa1: Etrada da fórmula F_2 e geração da fórmula nos formatos infixo e pós-fixo. Em seguida, é garada a árvoce da fórmula.

Fórmula de Entrada com Índices:

(|0 (|1 (|2 Socrates|3 ⊑|4 Homem|5)|6 ⊓|7 (|8 Homem|9 ⊑|10 Masculino|11)|12)|13 ⊨|14 (|15 Socrates|16 ⊑|17 Masculino|18)|19)|20

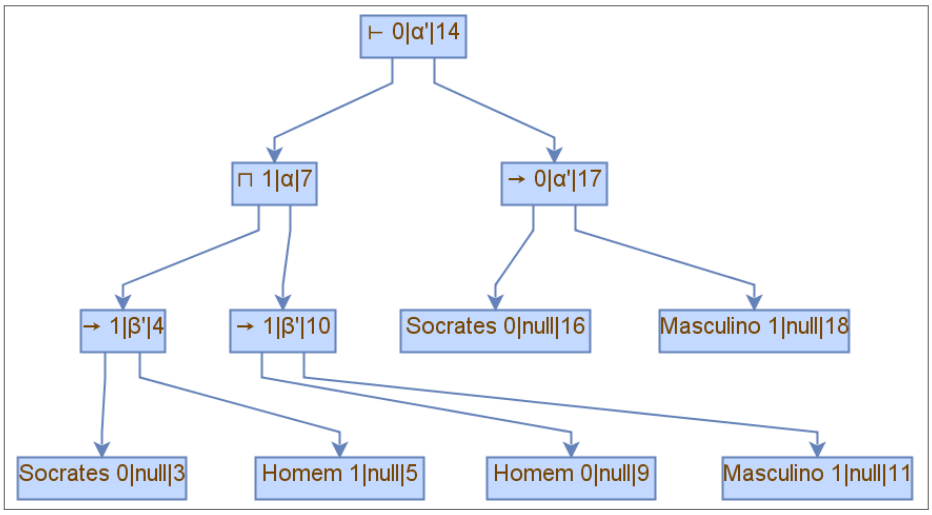
Fórmula Infixa:

(|0 (|1 (|2 Socrates|3 ⊑|4 Homem|5)|6 ⊓|7 (|8 Homem|9 ⊑|10 Masculino|11)|12)|13 ⊨|14 (|15 Socrates|16 ⊑|17 Masculino|18)|19)|20

Fórmula Pós-fixa:

Socrates|3 Homem|5 ⊑|4 Homem|9 Masculino|11 ⊑|10 ⊓|7 Socrates|16 Masculino|18 ⊑|17 ⊨|14

Figura 22 – Árvore de Prova de F_2



Fonte: O Autor (2019)

Etapa2: Entrada da matriz de prova para F_2 e atribuição dos elementos da matriz.

Matriz de Prova:
{{1,Socrates,0,null,5},{2,Homem,0,null,2},{3,Homem,1,null,1},{4,Masculino,0,null,4},
{5,Socrates,1,null,null},{6,Masculino,1,null,null}}

Matriz de Prova com Posições:
{{1,Socrates,0,3,5},{2,Homem,0,5,2},{3,Homem,1,9,1},{4,Masculino,0,11,4},
{5,Socrates,1,16,null},{6,Masculino,1,18,null}}

Etapa3: São obtidas as conexões, tabela 7, e a ordem de redução do sequente. Em seguida, é gerada a estrutura da prova parcial em sequentes, figura 23.

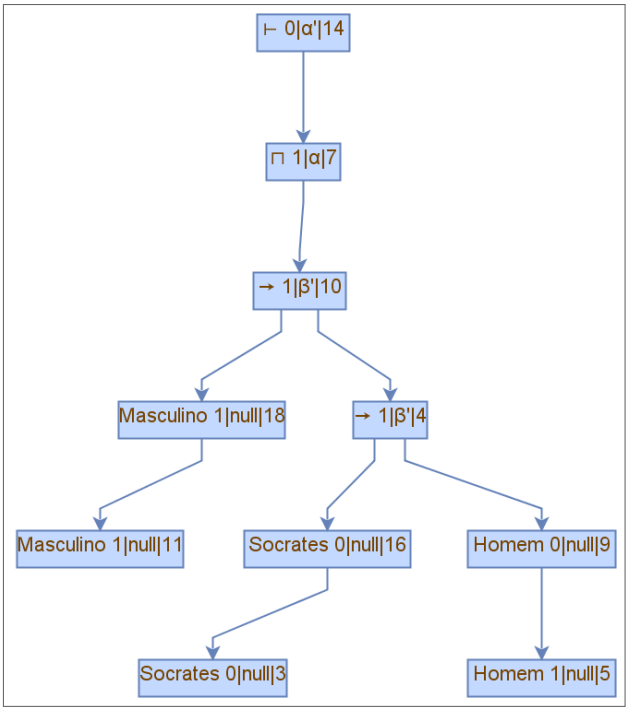
Tabela 7 – Conexões de Entrada

Conexão	Posição 1	Posição 2
1	18	11
2	16	3
3	9	11

Fonte: O Autor (2019)

OrdemReducao = {14,10,{18,11},4,{16,3},{9,11}};

Figura 23 – Estrutura Parcial da Prova em Sequentes



Fonte: O Autor (2019)

Etapa4: Geração do sequente a partir da estrutura parcial da prova, figura 23

(04) Socrates $\vdash$ Homem|| Homem $\vdash$ Homem
(03) (Socrates $\vdash$ Homem) || (Homem $\vdash$ Masculino)
(02) (((Socrates $\rightarrow$ Homem) , (Homem $\rightarrow$ Masculino)) $\vdash$ (Socrates $\rightarrow$ Masculino))
(01) (((Socrates $\rightarrow$ Homem) $\sqcap$ (Homem $\rightarrow$ Masculino)) $\vdash$ (Socrates $\rightarrow$ Masculino))

Exemplo3: Considere a fórmula F_3 como exemplo de entrada:

{ (Animal $\sqcap$ (($\exists$ haspart . Bone) $\sqsubseteq$ Vertebrate)) $\sqcap$ (Bird $\sqsubseteq$ (($\exists$ haspart . Bone) $\sqcap$ ($\exists$ haspart . Feather))) } $\models$ (Bird $\sqsubseteq$ Vertebrate)

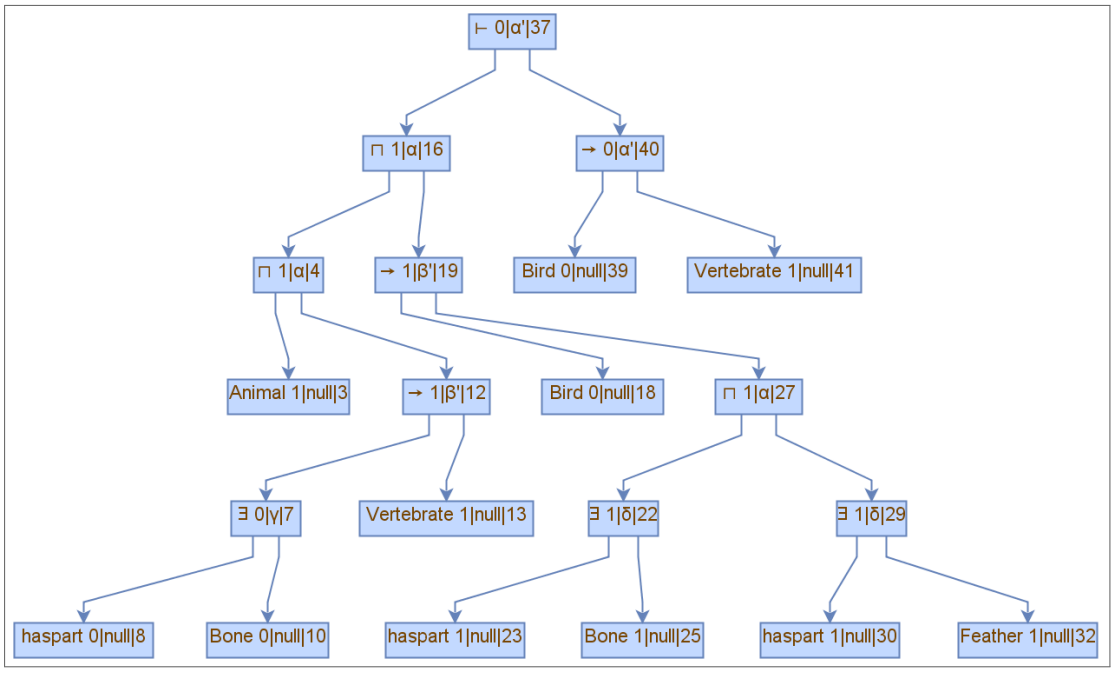
Etapa1: Etrada da fórmula F_3 e geração da fórmula nos formatos infixo e pós-fixo. Em seguida, é garada a árvoce da fórmula.

Fórmula de Entrada com Índices:
(|0 (|1 (|2 Animal|3 $\sqcap$|4 (|5 (|6 $\exists$|7 haspart|8 .|9 Bone|10)|11 $\sqsubseteq$|12 Vertebrate|13)|14)|15 $\sqcap$|16 (|17 Bird|18 $\sqsubseteq$|19 (|20 (|21 $\exists$|22 haspart|23 .|24 Bone|25)|26 $\sqcap$|27 (|28 $\exists$|29 haspart|30 .|31 Feather|32)|33)|34)|35)|36 $\models$|37 (|38 Bird|39 $\sqsubseteq$|40 Vertebrate|41)|42)|43

Fórmula Infixa:
(|0 (|1 (|2 Animal|3 $\sqcap$|4 (|5 (|6 haspart|8 $\exists$|7 Bone|10)|11 $\sqsubseteq$|12 Vertebrate|13)|14)|15 $\sqcap$|16 (|17 Bird|18 $\sqsubseteq$|19 (|20 (|21 haspart|23 $\exists$|22 Bone|25)|26 $\sqcap$|27 (|28 haspart|30 $\exists$|29 Feather|32)|33)|34)|35)|36 $\models$|37 (|38 Bird|39 $\sqsubseteq$|40 Vertebrate|41)|42)|43

Fórmula Pós-fixa:
Animal|3 haspart|8 Bone|10 $\exists$|7 Vertebrate|13 $\sqsubseteq$|12 $\sqcap$|4 Bird|18 haspart|23 Bone|25 $\exists$|22 haspart|30 Feather|32 $\exists$|29 $\sqcap$|27 $\sqsubseteq$|19 $\sqcap$|16 Bird|39 Vertebrate|41 $\sqsubseteq$|40 $\models$|37

Figura 24 – Árvore de Prova de F_3



Fonte: O Autor (2019)

Etapa2: Entrada da matriz de prova para F_3 e atribuição dos elementos da matriz.

Matriz de Prova:
{{1,Bird,0,null,1;2,Animal,1,null,2}, {3,Bird,0,null,3;4,hasPart,1,null,4},
{5,Bird,0,null,5;6,Bone,1,null,6}, {7,Bird,0,null,7;8,hasPart,1,null,8},
{9,Bird,0,null,9;10,Feather,1,null,10},
{11,Animal,0,null,11;12,hasPart,0,null,12;13,Bone,0,null,13;14,Vertebrate,1,null,14},
{15,Bird,1,null,15}, {16,Vertebrate,0,null,16}}.

Matriz de Prova com Posições:
{{1,Bird,0,18,1;2,Animal,1,3,2}, {3,Bird,0,39,3;4,hasPart,1,8,4},
{5,Bird,0,18,5;6,Bone,1,10,6}, {7,Bird,0,18,7;8,hasPart,1,23,8},
{9,Bird,0,18,9;10,Feather,1,32,10},
{11,Animal,0,3,11;12,hasPart,0,30,12;13,Bone,0,null,13;14,Vertebrate,1,13,14},
{15,Bird,1,39,15}, {16,Vertebrate,0,41,16}}.

Etapa3: São obtidas as conexões, tabela 8, e a ordem de redução do sequente. Em seguida, é gerada a estrutura da prova parcial em sequentes, figura 25.

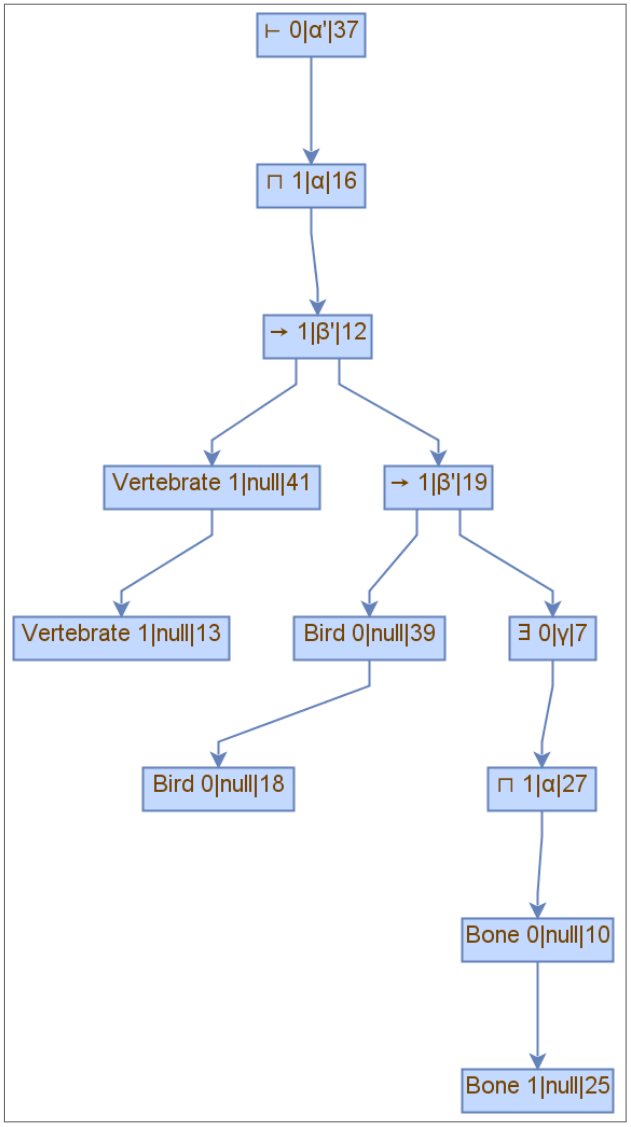
OrdemReducao = {37,12,{41,13},19,{39,18},7,27,{10, 25}};

Tabela 8 – Conexões de Entrada

Conexão	Posição 1	Posição 2
1	41	13
2	39	18
3	10	25

Fonte: O Autor (2019)

Figura 25 – Estrutura Parcial da Prova em Sequentes



Fonte: O Autor (2019)

Etapa4: Geração do sequente a partir da estrutura parcial da prova, figura 25

(06) Bone , Feather ⊢ Bone

(05) Bone ⊓ Feather ⊢ Bone

(04) (Bird ⊢ ((∃haspart.Bone) ⊓ (∃haspart.Feather))||((∃haspart.Bone) ⊓ (∃haspart.Feather))

$\vdash$ ($\exists$haspart.Bone))
(03) (Bird $\vdash$ ($\exists$haspart.Bone))||((($\exists$haspart.Bone) $\vdash$ Vertebrate)
(02) (((Animal $\sqcap$ (($\exists$haspart.Bone) $\rightarrow$ Vertebrate)) ,
(Bird $\rightarrow$ (($\exists$haspart.Bone) $\sqcap$ ($\exists$haspart.Feather)))) $\vdash$ (Bird $\rightarrow$ Vertebrate))
(01) (((Animal $\sqcap$ (($\exists$haspart.Bone) $\rightarrow$ Vertebrate)) $\sqcap$
(Bird $\rightarrow$ (($\exists$haspart.Bone) $\sqcap$ ($\exists$haspart.Feather)))) $\vdash$ (Bird $\rightarrow$ Vertebrate))

O processo de conversão aqui implementado e apresentado faz uso da lógica de descrições $\mathcal{ALC}$ com o uso do cálculo de sequentes para tradução da prova em uma estrutura de sequente. Outros cálculos e outras lógicas com outros poderes de expressividade podem ser formalizadas posteriormente. Os algoritmos estão disponíveis para download no seguinte repositório: https://AllisonMagno@bitbucket.org/AllisonMagno/convertesequentes.git.

5 CONCLUSÃO E TRABALHOS FUTUROS

Lógica de Descrições é um formalismo poderoso para representar conhecimento e realizar inferências, mas, dificilmente, encontramos especialistas em domínios que conseguem entender e usar a lógica. Boas soluções para inferências sobre bases de conhecimento em lógica de descrições já existem, porém, o formato das provas ainda é alvo de pesquisas quando se descreve os passos usados para se chegar às conclusões. Com essa ferramenta, será possível a leitura da sequência das provas de forma mais compreensível e de um entendimento mais sequencial apresentado como uma prova em sequentes.

O usuário final pode utilizar esse sistema apenas como um detentor de conhecimento do domínio em que é especialista, sem, necessariamente, dominar ou deter o conhecimento técnico sobre a lógica de descrições. O uso dessa ferramenta busca facilitar o entendimento sobre as provas geradas e sobre a descrição dos passos que foram utilizados para a construção da prova. Contudo, é possível uma maior amplitude no conjunto de usuários que podem ser auxiliados no processo de apoio à decisão, sobretudo, em problemas complexos, vendo que, a saída com o resultado e todos os passos da prova estão disponíveis para o usuário no cálculo de sequentes, que apresenta uma linguagem um pouco mais amigável e mais simples para os usuários.

5.1 CONTRIBUIÇÕES

O sistema, aqui apresentado, tem como objetivo dar continuidade ao trabalho de Palmeira (2017), implementando suas definições de algoritmos e adicionando outros para melhorar o processo de codificação do trabalho. Ele vai auxiliar usuários finais que não têm o domínio da lógica. Dificilmente, encontramos especialistas em domínios que conseguem entender e usar a lógica. Com essa ferramenta, será possível a leitura da sequência das provas de forma mais próxima a linguagem humana.

5.2 TRABALHOS FUTUROS

Esse trabalho pode ser ampliado em:

- Implementar a função para gerar a ordem de redução dos sequentes em $\mathcal{ALC}$;
- Utilizar outras abordagens do cálculo de sequentes;
- Usar de forma prática para operações com raciocínio automático com DL;
- Converter a saída no Cálculo de Sequentes para uma saísa em linguagem natural;
- Aumentar o poder de expressividade da lógica utilizada.

- Em outro trabalho, adicionar uma etapa de tradução do cálculo de sequentes para texto, utilizando técnicas de Processamento de Linguagem Natural.

REFERÊNCIAS

BAADER, F.; CALVANESE, D.; MCGUINNESS, D. L.; NARDI, D.; PATEL-SCHNEIDER, P. F. The description logic handbook: Theory, implementation, and applications. In: . [S.l.: s.n.], 2003.

BAADER, F.; HORROCKS, I.; SATTLER, U. In: *Description Logics.* [S.l.: s.n.], 2008. p. 135–179.

BIBEL, W. Automated theorem proving. In: *Automated theorem proving.* [S.l.: s.n.], 1987.

BORGIDA, A.; FRANCONI, E.; HORROCKS, I. Explaining alc subsumption. In: *European Conference on Artificial Intelligence.* [S.l.: s.n.], 2000.

BORGIDA, A.; SERAFINI, L. Distributed description logics: Assimilating information from peer sources. In: *Journal on data semantics.* [S.l.: s.n.], 2003. p. 153–184.

ENDERTON, H.; ENDERTON, H. B. In: *Mathematical Introduction to Logic.* [S.l.: s.n.], 2001.

FREITAS, F.; JR, Z. C.; STUCKENSCHMIDT, H. Towards checking laws' consistency through ontology design: The case of brazilian vehicles' laws. In: *Journal of Theoretical and Applied Electronic Commerce Research.* [S.l.: s.n.], 2011. p. 112–126.

FREITAS, F.; OTTEN, J. A connection calculus for the description logic alc. In: . [S.l.: s.n.], 2016.

FREITAS, F.; VARZINCZAK2, I. Cardinality restrictions within description logic connection calculi. In: . [S.l.: s.n.], 2018. p. 226–241.

GENTZEN, G. In: *Untersuchungen über das logische Schliessen. Mathematische zeitschrift.* [S.l.: s.n.], 1935. p. 176–210.

GIRARD, J.-Y.; TAYLOR, P.; LAFONT, Y. *Proofs and Types.* [S.l.]: Cambridge University Press, 1989. ISBN 0-521-37181-3.

HAMMACK, R. In: *Book of Proof.* [S.l.: s.n.], 2009.

HAN, S.; HUTTER, A.; STECHELE, W. A reasoning approach to enable abductive semantic explanation upon collected observations for forensic visual surveillance. In: *IEEE International Conference.* [S.l.: s.n.], 2011.

KFOURY, A.; MOLL, R. N.; ARBIB, M. A. A programming approach to computability. In: *Springer Science Business Media.* [S.l.: s.n.], 2012.

MANFRED, S.; SMOLKA, G. Attributive concept descriptions with complements. artificial intelligence. In: . [S.l.: s.n.], 1991. p. 1–126.

NOY, N. F.; CORONADO, S. de; SOLBRIG, H.; FRAGOSO, G.; HARTEL, F. W.; MUSEN, M. A. Representing the nci thesaurus in owl dl: Modeling tools help modeling languages. applied ontology. In: . [S.l.: s.n.], 2008. p. 173–190.

OTTEN, J. A non-clausal connection calculus. in: Automated reasoning with analytic tableaux and related methods. In: *Springer*. [S.l.: s.n.], 2011. p. 226–241.

OTTEN, J.; KREITZ, C. A connection based proof method for intuitionistic logic. in: Theorem proving with analytic tableaux and related methods. In: . [S.l.: s.n.], 1995. p. 122–137.

PALMEIRA, E. S. Conversão de provas em lógica de descrições alc geradas pelo método de conexões para sequentes. In: . [S.l.: s.n.], 2017.

RIBEIRO, M. M. Belief revision in non-classical logics. In: . [S.l.: s.n.], 2012.

ROSEN, K. H. In: *Handbook of Discrete and Combinatorial Mathematics*. [S.l.: s.n.], 1999.

APÊNDICE A – CÓDIGO JAVA DO PROCESSO DE CONVERSÃO

Listing A.1 – Código em linguagem de programação Java dos algoritmos utilizados no processo de conversão

```
1
//* CONVERSAO DE PROVAS EM LOGICA DE DESCRICOES EM ALC GERADAS
3  PELO METODO DAS CONEXOES PARA SEQUENTES */

5 package convertesequentes;

7 public class Literal {
      String rotulo;
9     int posicao;

11    public Literal()
      {
13
      }
15 }

17 package convertesequentes;

19 public class NoArvore {

21    public String rotulo;
      public int polaridade;
23    public int posicao;
      public String tipo;
25    public String[] posSubst = new String [2];
      public String instancia;
27    public NoArvore direita;
      public NoArvore esquerda;
29
      public NoArvore()
31    {

33    }

35 }//Fim NoArvore

37 package convertesequentes;

39 public class Elemento {

41    public int id;
      public String literal;
43    public int polaridade;
      public int posicao;
45    public Conexao conexoes[];

47    public Elemento()
      {
49
```

```
    }

}

package convertesequentes;

public class NoF {
    Literal[] No;
    NoF direita;
    NoF esquerda;

    public NoF()
    {

    }
}

package convertesequentes;

public class Conexao {

    int idelemento0;
    int idelemento1;
    int ordem;

    public Conexao()
    {

    }

}

package convertesequentes;

public class ConexaoNo {

    int posicao1;
    int posicao2;
    int ordem;

    public ConexaoNo()
    {

    }
}

package convertesequentes;
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */

import org.jgrapht.graph.DefaultEdge;

/**
 *
```

```
107  * @author Allis
     */
109 public class EdgeConv {

111     private String label;

113     /**
         * Constructs a relationship edge
115      *
         * @param label the label of the new edge.
117      *
         */
119     public EdgeConv(String label)
        {
121         this.label = label;
        }
123
        /**
125      * Gets the label associated with this edge.
         *
127      * @return edge label
         */
129     public String getLabel()
        {
131         return label;
        }
133
        @Override
135     public String toString()
        {
137         return label;
        }
139 }

141 package convertesequentes;

143 import com.mxgraph.layout.hierarchical.mxHierarchicalLayout;
    import com.mxgraph.util.mxCellRenderer;
145 import static com.sun.javafx.fxml.expression.Expression.add;
    import java.awt.BorderLayout;
147 import java.awt.Color;
    import java.awt.Container;
149 import java.awt.Graphics2D;
    import java.awt.Image;
151 import java.awt.image.BufferedImage;
    import java.io.BufferedReader;
153 import java.io.File;
    import java.io.FileInputStream;
155 import java.io.FileNotFoundException;
    import java.io.FileReader;
157 import java.io.FileWriter;
    import java.io.IOException;
159 import java.io.InputStreamReader;
    import java.io.PrintWriter;
161 import java.io.UnsupportedEncodingException;
    import java.net.URL;
163 import java.util.ArrayList;
```

```
import java.util.NoSuchElementException;
import java.util.Stack;
import javax.imageio.ImageIO;
import javax.swing.Icon;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import org.jgrapht.ext.JGraphXAdapter;
import org.jgrapht.graph.SimpleGraph;

public class ConverteSequentes {
    public static String cstr1 = "   ";
    public static String cstr2 = "   ";
    public static String cstr3 = "   ";
    public static String cstr4 = "   ";
    public static String cstr5 = "  ";
    public static String cstr6 = "   ";
    public static String cstr7 = "   ";
    public static int MAXTAMFOR = 55;
    public static int pos[];
    public static File imgFile = new File(".\\Dados\\Arvores\\arvoredeprova.png");
    public static File imgFileEstParcial = new File(".\\Dados\\Arvores\\
        estruturaparcial.png");
    public static SimpleGraph<String, EdgeConv> g1  = new SimpleGraph<>(EdgeConv.
        class);
    public static SimpleGraph<String, EdgeConv> g2  = new SimpleGraph<>(EdgeConv.
        class);
    public static Stack S = null;
    public static NoF AuxNoF = null;
    public static String ladoEstrutura = "";


    public static void main(String[] args) throws FileNotFoundException, IOException
    {
        String path = ".\\Dados\\Entrada01\\Entrada.txt";
        String pathElementosF = ".\\Dados\\Entrada01\\ElementosF[].txt";
        String VetorConexoes = ".\\Dados\\Entrada01\\VetorConexoes.txt";
        String EntradaMatrizProva = ".\\Dados\\Entrada01\\EntradaMatrizProva.txt";
        String[] Entrada01C = {"33","36","12","9","{37,10}","15","{14,35}","23","18"
            ,"{7,28}"};
        String[] Entrada02C = {"26","28","31","{37,10}","15","{14,35}","23","18","
            {7,28}"};
        String[] Entrada03C = {"33","36","12","9","{37,10}","15","{14,35}","23","18"
            ,"{7,28}"};
        String[] EntradaC = Entrada01C;
        Literal auxF[] = new Literal[MAXTAMFOR];

        //CARREGANDO A F RMULA
        System.out.println("###ROTULO E POSICAO EM F[]###");
        auxF = CreateF(path, pathElementosF, MAXTAMFOR);
        //Verifica o tamanho de F
        int contF = 0;
        for(int k = 0;k < auxF.length;){
            if (auxF[k] == null){
                k++;
```

```
215             }else{
                    contF++;
217                 k++;
                }
219         }
            //Transfere os valores de auxF para F com o tamanho correto
221         Literal F[] = new Literal[contF];
            for(int k = 0; k < F.length;)
223         {
                F[k] = auxF[k];
225             k++;
            }
227         //Escrevendo F
            for(int i = 0; i < F.length; i++)
229         {
                if(F[i] != null)
231                 System.out.print("" + F[i].rotulo + "_" + "" + F[i].posicao + " ");
            }
233         //FIM CARREGANDO A F RMULA

235         //CONVERTENDO PARA Fin
            System.out.println("\n\n###ROTULO E POSICAO EM Fin[]###");
237         Literal auxFin[] = new Literal[F.length];
            auxFin = converteEmInFixa(F);
239         //Verifica o tamanho de Fin
            int contFin = 0;
241         for(int k = 0;k < auxFin.length;){
                if (auxFin[k] == null){
243                 k++;
                }else{
245                 contFin++;
                    k++;
247             }
            }
249         //Transfere os valores de auxFin para Fin com o tamanho correto
            Literal Fin[] = new Literal[contFin];
251         for(int k = 0; k < Fin.length;)
            {
253             Fin[k] = auxFin[k];
                k++;
255         }
            //Escrevendo Fin
257         for(int i = 0; i < Fin.length; i++)
            {
259             if(Fin[i] != null)
                System.out.print("" + Fin[i].rotulo + "_" + "" + Fin[i].posicao + " ");
261         }
            //FIM CONVERTENDO PARA Fin
263
            //CONVERTENDO PARA Fp
265         System.out.println("\n\n###ROTULO E POSICAO EM Fp[]###");
            Literal auxFp[] = new Literal[Fin.length];
267         auxFp = converteEmPosFixa(Fin);
            //Verifica o tamanho de Fp
269         int contFp = 0;
            for(int k = 0;k < auxFp.length;){
271             if (auxFp[k] == null){
```

```
                k++;
273             }else{
                    contFp++;
275                 k++;
                }
277         }
            //Transfere os valores de auxFp para Fp com o tamanho correto
279         Literal Fp[] = new Literal[contFp];
            for(int k = 0; k < Fp.length;)
281         {
                Fp[k] = auxFp[k];
283             k++;
            }
285         //Escrevendo Fp
            for(int i = 0; i < Fp.length; i++)
287         {
                if(Fp[i] != null)
289                 System.out.print("" + Fp[i].rotulo + "_" + "" + Fp[i].posicao + " ")
                        ;
            }
291         System.out.print("\n");
            //FIM CONVERTENDO PARA Fp
293
            //CONSTROI ARVORE DE FORMULA
295         String arcos[] = new String[4];
            NoArvore ArvoreProva = constroiArvore(Fp, "0", arcos, 0, 0);
297         imprimirArvoreProva(ArvoreProva, true);
            //openImage(".\\Dados\\Arvores\\arvoredeprova.png");
299         //FIM CONSTROI ARVORE DE FORMULA

301         //Carrega Vetor C
            NoArvore[][] auxC;
303         auxC = carregaVetorC(EntradaC, ArvoreProva);
            NoArvore[] C = new NoArvore[auxC.length];
305         for(int i = 0; i < C.length; i++)
            {
307             if(auxC[i][1] != null){
                    C[i] = auxC[i][0];
309                 C[i].direita = auxC[i][1];
                    C[i].esquerda = null;
311
                }
313             else
                {
315                 C[i] = auxC[i][0];
                    C[i].direita = null;
317                 C[i].esquerda = null;
                }
319         }
            //Fim Carrega Vetor C
321
            //Controi EstruturaParcial
323         NoArvore NoEst = new NoArvore();
            NoEst = constroiEstruturaSequentes(C, 0, true);
325         imprimirArvoreReducao(NoEst, true);
            //openImage(".\\Dados\\Arvores\\estruturaparcial.png");
327         //Fim Constroi EstruturaParcial
```

```
329         //Mostra Vetor de Conex es
            System.out.print("\n###CONEX ES###\n");
331         Show(VetorConexoes);

333         //Mostra Matriz N o-Clausal Simplificada
            System.out.print("\n###MATRIZ###\n");
335         Show(EntradaMatrizProva);

337         //Constroi Sequente
            System.out.print("\n###SEQUENTE###\n");
339         NoF Fno = new NoF();
            Fno.No = F;
341         EscreveNoF(Fno);
            //NoF Sequente = constroiSequentes(Fno, ArvoreProva, NoEst, 'a', "", 0, S);
343         //for(int i = 0; i < C.length; i++){
            AuxNoF = Fno;
345         NoF Sequente2 = constroiSequentes(Fno, ArvoreProva, NoEst, 'a', "", 0, S);
            //EscreveNoF(Sequente2);
347         //}

349         //Fim Constroi Sequente
            //String teste = "";
351     }//Fim main

353     public static void Show (String path) throws IOException
        {
355         // abertura(leitura) do arquivo
            BufferedReader buffRead = new BufferedReader(new FileReader(path));
357         // loop que l   e imprime todas as linhas do arquivo
            String linha = buffRead.readLine();
359         //Escrevendo linha na tela
            while (linha != null) {
361                 System.out.println(linha);
                    linha = buffRead.readLine();
363         }//Fim While 1

365     }//Fim Show

367     public static void EscreveNoF (NoF Fno) throws IOException
        {
369         System.out.print("\n");
            if(Fno.direita != null)
371         {
                for(int i = 0; i < Fno.direita.No.length; i++){
373                 System.out.print(Fno.direita.No[i].rotulo + " ");
                }
375             if(Fno.esquerda != null)
                {
377                 for(int i = 0; i < Fno.esquerda.No.length; i++){
                        System.out.print(Fno.esquerda.No[i].rotulo + " ");
379                 }
                }
381         }
            else
383         {
                for(int i = 0; i < Fno.No.length; i++){
```

```
385                 if(Fno.No[i] != null)
                        System.out.print(Fno.No[i].rotulo + " ");
387             }
            }
389
        }//Fim Show
391
        public static Literal[] CreateF (String path, String pathElementosF, int tamF)
            throws FileNotFoundException, IOException
393     {
            Literal F[] = new Literal[tamF];
395         FileReader j = new FileReader(path);
            String path2 = pathElementosF;
397         FileWriter o = new FileWriter(path2);
            BufferedReader br = new BufferedReader(j);
399         PrintWriter out = new PrintWriter(o);
            String linha;
401         int i = 0;
            while ((linha = br.readLine()) != null){
403             linha = linha.replace(" ","\n");
                out.print(linha);
405         }  //FIM WHILE

407         out.flush();
            j.close();
409         o.close();

411         FileReader j2 = new FileReader(pathElementosF);
            BufferedReader br2 = new BufferedReader(new InputStreamReader(new
                FileInputStream(pathElementosF), "UTF-8"));
413         String linha2;
            int i2 = 0;
415         while ((linha2 = br2.readLine()) != null){
                F[i2] = new Literal();
417             F[i2].rotulo = (String)linha2;
                F[i2].posicao = i2;
419             i2++;
            }  //FIM WHILE
421         out.flush();
            j2.close();
423
            //Elimina os valores null para inserir em auxFin
425         int cont = 0;
            Literal auxF[] = new Literal[F.length];
427         for(int k = 0;k < F.length;){

429             if (F[k] == null){
                    k++;
431             }else{
                    auxF[cont] = F[k];
433                 cont++;
                    k++;
435             }
            }
437         return auxF;

439     }//Fim CreateF
```

```
441     public static Literal[] converteEmInFixa(Literal F[])
        {
443         Literal Fin[] = new Literal[F.length];
            Literal quant = null;
445         int i = 0;
            for(int m = 0; m <F.length;)
447         {
                if(F[m] != null)
449             {
                    if(!F[m].rotulo.equals("   ") && !F[m].rotulo.equals("   ") && !F[m
                        ].rotulo.equals("."))
451                 {
                        Fin[m] = F[m];
453                     m = m + 1;
                    }
455                 else
                    {
457                     if(F[m].rotulo.equals("   ") || F[m].rotulo.equals("   "))
                        {
459                         quant = F[m];
                            m = m + 1;
461                     }
                        else
463                     {
                            if(F[m].rotulo.equals("."))
465                         {
                                Fin[m] = quant;
467                             m = m +1;
                            }
469                     }
                    }
471             }
                else
473             {
                    m++;
475             }
            }
477         //Elimina os valores null para inserir em auxFin
            int cont = 0;
479         Literal auxFin[] = new Literal[F.length];
            for(int k = 0;k < Fin.length;){
481
                if (Fin[k] == null){
483                 k++;
                }else{
485                 auxFin[cont] = Fin[k];
                    cont++;
487                 k++;
                }
489         }

491         return auxFin;
        }//Fim converteEmInFixa
493
        public static Literal[] converteEmPosFixa(Literal Fin[])
495     {
```

```
        Literal Fp[] = new Literal[Fin.length];
497     //Stack Pilha = new Stack();
        Stack PilhaFp = new Stack();
499     Literal lastelement = null;
        int i = 0;
501     int j = 0;
        int ultimoi =0;
503     Fin[0].rotulo = "(";
        for(int m = 0; m <Fin.length; m++)
505     {
            if(Fin[m] != null)
507         {
                if(Fin[m].rotulo.equals("("))
509             {
                    PilhaFp.push(Fin[m]);
511             }
                else
513             {
                    if(!Fin[m].rotulo.equals(cstr1) & !Fin[m].rotulo.equals(cstr2) &
                         !Fin[m].rotulo.equals(cstr3) & !Fin[m].rotulo.equals(cstr4)
                         & !Fin[m].rotulo.equals(cstr5) & !Fin[m].rotulo.equals(
                        cstr6) & !Fin[m].rotulo.equals(cstr7) & !Fin[m].rotulo.
                        equals(")"))
515                 {
                        Fp[j] = Fin[m];
517                     j++;
                        //PilhaFp.push(Fin[m]);
519                 }
                    else
521                 {
                        if(Fin[m].rotulo.equals(cstr1) || Fin[m].rotulo.equals(cstr2
                            ) || Fin[m].rotulo.equals(cstr3) || Fin[m].rotulo.equals
                            (cstr4) || Fin[m].rotulo.equals(cstr5) || Fin[m].rotulo.
                            equals(cstr6) || Fin[m].rotulo.equals(cstr7))
523                     {
                            PilhaFp.push(Fin[m]);
525                     }
                        else
527                     {
                            if(Fin[m].rotulo.equals(")"))
529                         {
                                i = m;
531                             if (PilhaFp.empty() != true)
                                {
533                                 lastelement = (Literal) PilhaFp.lastElement();
                                }
535                             while((PilhaFp.empty() != true) && (!lastelement.
                                rotulo.equals("(")))
                                {
537                                 if(lastelement.rotulo.equals(cstr1) ||
                                        lastelement.rotulo.equals(cstr2) ||
                                        lastelement.rotulo.equals(cstr3) ||
                                        lastelement.rotulo.equals(cstr4) ||
                                        lastelement.rotulo.equals(cstr5) ||
                                        lastelement.rotulo.equals(cstr6) ||
                                        lastelement.rotulo.equals(cstr7))
                                    {
```

```
539                                     //PilhaFp.push(Pilha.lastElement());
                                        if (ultimoi > i)
541                                     {
                                            Fp[j] = (Literal) PilhaFp.lastElement();
543                                         j++;
                                            PilhaFp.pop();
545                                         if (PilhaFp.empty() != true){
                                                lastelement = (Literal) PilhaFp.
                                                    lastElement();
547                                         }
                                        }
549                                     else
                                        {
551                                         Fp[j] = (Literal) PilhaFp.lastElement();
                                            j++;
553                                         PilhaFp.pop();
                                            if (PilhaFp.empty() != true)
555                                         {
                                                lastelement = (Literal) PilhaFp.
                                                    lastElement();
557                                         }
                                        }
559                                 }
                                    //Pilha.pop();
561                             }
                                if (PilhaFp.empty() != true)
563                             {
                                    PilhaFp.pop();
565                             }
                                if (PilhaFp.empty() != true)
567                             {
                                    lastelement = (Literal) PilhaFp.lastElement();
569                             }
                            }
571                     }
                    }
573             }
            }
575     }

577     return Fp;
    }//Fim converteEmPosFixa
579
    public static String[] buscaTipoPol(String Rotulo, String pol)
581 {
        String btPol[][] = new String[13][5];
583     String[] trl = new String[3];

585     if(Rotulo.equals("   ") && pol.equals("1"))
        { trl[0] = "  "; trl[1] = "1"; trl[2] = "1"; return trl; }
587     else if(Rotulo.equals("   ") && pol.equals("0"))
        { trl[0] = "  "; trl[1] = "0"; trl[2] = "0"; return trl; }
589     else if(Rotulo.equals("  ") && pol.equals("1"))
        { trl[0] = "  "; trl[1] = "0"; trl[2] = ""; return trl; }
591     else if(Rotulo.equals("  ") && pol.equals("0"))
        { trl[0] = "  "; trl[1] = "1"; trl[2] = ""; return trl; }
593     else if(Rotulo.equals("   ") && pol.equals("0"))
```

```
        { trl[0] = "  '"; trl[1] = "1"; trl[2] = ""; return trl; }
595     else if(Rotulo.equals("   ") && pol.equals("0"))
        { trl[0] = "  '"; trl[1] = "0"; trl[2] = "1"; return trl; }
597     else if(Rotulo.equals("   ") && pol.equals("1"))
        { trl[0] = "  '"; trl[1] = "1"; trl[2] = "0"; return trl; }
599     else if(Rotulo.equals("   ") && pol.equals("0"))
        { trl[0] = "  "; trl[1] = "0"; trl[2] = "0"; return trl; }
601     else if(Rotulo.equals("   ") && pol.equals("1"))
        { trl[0] = "  "; trl[1] = "1"; trl[2] = "1"; return trl; }
603     else if(Rotulo.equals("   ") && pol.equals("0"))
        { trl[0] = "  "; trl[1] = "0"; trl[2] = "1"; return trl; }
605     else if(Rotulo.equals("   ") && pol.equals("1"))
        { trl[0] = "  "; trl[1] = "1"; trl[2] = "1"; return trl; }
607     else if(Rotulo.equals("   ") && pol.equals("1"))
        { trl[0] = "  "; trl[1] = "1"; trl[2] = "0"; return trl; }
609     else if(Rotulo.equals("   ") && pol.equals("0"))
        { trl[0] = "  "; trl[1] = "0"; trl[2] = "0"; return trl; }
611
        return trl;
613 }//Fim buscaTipoPol

615 public static NoArvore atualizaTipoPol(NoArvore No, boolean PrimeiroNo){
        //Busca tipo e polaridade
617     String[] trl = new String[3];
        if(PrimeiroNo == true)
619     {
            No.polaridade = 0;
621         trl = buscaTipoPol(No.rotulo, "0");
        }
623     else
            trl = buscaTipoPol(No.rotulo, Integer.toString(No.polaridade));
625
        No.tipo = trl[0];
627     if(trl[1] != "" && trl[1] != null && No.esquerda != null)
            No.esquerda.polaridade = Integer.parseInt(trl[1]);
629     if(trl[2] != "" && trl[2] != null && No.direita != null)
            No.direita.polaridade = Integer.parseInt(trl[2]);
631     if(No.direita != null)
            atualizaTipoPol(No.direita, false);
633     if(No.esquerda != null)
            atualizaTipoPol(No.esquerda, false);
635
        return No;
637
    }//Fim atualizaTipoPol
639
    public static void atualizaPosicao(Literal F[])
641 {
        int idx = 1;
643     int j = 0;
        F[0].rotulo = "(";
645     pos = new int[F.length];
        for(int m = 0; m < F.length; m++)
647     {
            if(!F[m].rotulo.equals("(") && !F[m].rotulo.equals(")"))
649         {
                idx = idx + 1;
```

```
651                 j = 0;
            }
653             while(j < pos.length - 1)
            {
655                 j = j + 1;
                if(pos[j] == m)
657                 {
                    pos[j] = idx - 1;
659                 j = pos.length;
                }
661             }
        }
663     }//Fim atualizaPosicaoOld

665     public static NoArvore constroiArvore(Literal[] Fp, String pol, String[] arcos,
            int posBetal, int posGDelta) throws IOException, NoSuchElementException
        {
667         NoArvore no = null;
            String[] trl = new String[3];
669         int posstr;
            int posBl = posBetal;
671         int posGD = posGDelta;

673         Stack stack = new Stack();
            for(int i = 0; i < Fp.length ; i++)
675         {
                if(!Fp[i].rotulo.equals(cstr1) & !Fp[i].rotulo.equals(cstr2) & !Fp[i].
                    rotulo.equals(cstr3) & !Fp[i].rotulo.equals(cstr4) & !Fp[i].rotulo.
                    equals(cstr5) & !Fp[i].rotulo.equals(cstr6) & !Fp[i].rotulo.equals(
                    cstr7) & !Fp[i].rotulo.equals(")"))
677             {
                    no = new NoArvore();
679                 no.rotulo = Fp[i].rotulo;
                    if(pol != "" && pol != null)
681                     no.polaridade = Integer.parseInt(pol);
                    //if(pol.equals(""))
683                     //no.polaridade = null;
                    no.posicao = Fp[i].posicao;
685                 no.tipo = "";
                    no.posSubst[0] = Integer.toString(posBl);
687                 no.posSubst[1] = Integer.toString(posGD);
                    no.instancia = "";
689                 stack.push(no);
                }
691             else
                {
693                 trl = buscaTipoPol(Fp[i].rotulo, (String)pol);
                    if (trl[0].equals("  "))
695                 {
                        if(arcos[0] != "" && arcos[1] != "" && arcos[0] != null && arcos
                            [1] != null)
697                     {
                            arcos[0] = Integer.toString((Integer.parseInt(arcos[0]) + 1)
                                );
699                         arcos[1] = Integer.toString((Integer.parseInt(arcos[0]) + 1)
                                );
                        }
```

```
701                     //no = new NoArvore(Fp[index].rotulo,Fp[index].posicao,pol, trl
                            [0],arcos[0], arcos[1]);
                        no = new NoArvore();
703                     no.rotulo = Fp[i].rotulo;
                        if(pol != "" && pol != null)
705                         no.polaridade = Integer.parseInt(pol);
                        no.posicao = Fp[i].posicao;
707                     no.tipo = trl[0];
                        no.posSubst[0] = arcos[0];
709                     no.posSubst[1] = arcos[1];
                        no.instancia = "";
711                 }
                    else if (trl[0].equals("  '"))
713                 {
                        if(arcos[0] != "" && arcos[1] != "" && arcos[0] != null && arcos
                            [1] != null)
715                     {
                            arcos[2] = Integer.toString((Integer.parseInt(arcos[2]) + 1)
                                );
717                         arcos[3] = Integer.toString((Integer.parseInt(arcos[2]) + 1)
                                );
                        }
719                     posBl = Fp[i].posicao;
                        //no = new NoArvore(Fp[index].rotulo, Fp[index].posicao, pol,
                            trl[0],arcos[2], arcos[3]);
721                     no = new NoArvore();
                        no.rotulo = Fp[i].rotulo;
723                     if(pol != "" && pol != null)
                            no.polaridade = Integer.parseInt(pol);
725                     no.posicao = Fp[i].posicao;
                        no.tipo = trl[0];
727                     no.posSubst[0] = arcos[2];
                        no.posSubst[1] = arcos[3];
729                     no.instancia = "";
                    }
731                 else if((trl[0].equals("  ")) || (trl[0].equals("  ")))
                    {
733                     posGD = Fp[i].posicao;
                        //no = new NoArvore(Fp[index].rotulo, Fp[index].posicao, pol,
                            trl[0]);
735                     no = new NoArvore();
                        no.rotulo = Fp[i].rotulo;
737                     if(pol != "" && pol != null)
                            no.polaridade = Integer.parseInt(pol);
739                     no.posicao = Fp[i].posicao;
                        no.tipo = trl[0];
741                     no.posSubst[0] = "";
                        no.posSubst[1] = "";
743                     no.instancia = "";
                    }
745                 else
                    {
747                     //no = new NoArvore(Fp[index].rotulo, Fp[index].posicao, pol,
                            trl[0]);
                        no = new NoArvore();
749                     no.rotulo = Fp[i].rotulo;
                        if(pol != "" && pol != null)
```

```
751                         no.polaridade = Integer.parseInt(pol);
                        no.posicao = Fp[i].posicao;
753                     no.tipo = trl[0];
                        no.posSubst[0] = "";
755                     no.posSubst[1] = "";
                        no.instancia = "";
757
                    }
759                 if(stack.empty() == false)
                        no.esquerda = (NoArvore)stack.pop();
761                 //no.esquerda.polaridade = Integer.parseInt(trl[2]);
                    if(stack.empty() == false)
763                     no.direita = (NoArvore)stack.pop();
                    //no.esquerda.polaridade = Integer.parseInt(trl[1]);
765                 stack.push(no);
                }
767         }
            no = atualizaTipoPol(no, true);
769
            return no;
771     }//Fim constroiArvore

773     public static Elemento[] atribuiPosicao(Elemento[] matriz, Literal[] Fp, int
            indice)
        {
775         boolean achou = false;

777         for(int i = 0; i < matriz.length; i++)
            {
779             if(matriz[i] instanceof Elemento == false)
                    atribuiPosicao(matriz, Fp, indice);
781             else
                {
783                 //   um elemento
                    do{
785                     if(Fp[indice].rotulo.equals(matriz[i].literal))
                        {
787                         matriz[i].posicao = Fp[indice].posicao;
                            achou = true;
789                         indice = indice + 1;
                        }
791                     else
                        {
793                         indice = indice + 1;
                        }
795                 }while((indice > Fp.length) || (achou == true));
                }
797         }
            return matriz;
799     }//Fim atribuiPosicao

801     public static NoArvore buscaPosicaoNoArvore(String posicao, NoArvore No) throws
            NumberFormatException
        {
803         NoArvore NoEst = new NoArvore();

805         if(No != null && (Integer.toString(No.posicao)).equals(posicao))
```

```
        {
807         NoEst = No;
            return NoEst;
809     }
        if(No.esquerda == null && No.direita == null){
811         return null;
        }
813     NoEst = buscaPosicaoNoArvore(posicao, No.direita);
        if(NoEst != null && (Integer.toString(NoEst.posicao)).equals(posicao)){
815         return NoEst;
        }
817
        NoEst = buscaPosicaoNoArvore(posicao, No.esquerda);
819     return NoEst;

821 }//Fim buscaPosicaoNoArvore

823 public static NoArvore[][] carregaVetorC(String[] strC, NoArvore No)  throws
        FileNotFoundException, IOException
    {
825     NoArvore C[][] = new NoArvore[strC.length][2];
        String NoDuplo[];
827     for(int k = 0; k < strC.length; k++)
        {
829         if(strC[k].startsWith("{"))
            {
831             NoDuplo = new String[2];
                NoDuplo = strC[k].replace("{", "").replace("}", "").split(",");
833             for(int m = 0; m < 2;)
                {
835                 C[k][m] = buscaPosicaoNoArvore(NoDuplo[m], No);
                    m++;
837             }
            }
839         else
            {
841             C[k][0] = buscaPosicaoNoArvore(strC[k], No);
            }
843     }

845     return C;
    }//Fim carregaVetorC
847
    public static NoArvore constroiEstruturaSequentes(NoArvore C[], int idx, boolean
         primeiro)
849 {
        int i = idx;
851     NoArvore noEst = null;
        String temp;
853
        if(!C[i].rotulo.equals(cstr1) && !C[i].rotulo.equals(cstr2) &&
855         !C[i].rotulo.equals(cstr3) && !C[i].rotulo.equals(cstr4) &&
            !C[i].rotulo.equals(cstr5) && !C[i].rotulo.equals(cstr6) &&
857         !C[i].rotulo.equals(cstr7))
        {
859         return C[i];
        }
```

```
861         else
            {
863             noEst = C[i];
                if(noEst.direita == null)
865                 noEst.direita = constroiEstruturaSequentes(C, i + 1, false);

867             if(noEst.tipo.equals("  ") || noEst.tipo.equals("  '"))
                {
869                 if(noEst.esquerda == null)
                        noEst.esquerda = constroiEstruturaSequentes(C, i + 2, false);
871             }
            }
873         return noEst;
        }//Fim constroiEstruturaSequentes
875
        public static NoF constroiSequentes(NoF F, NoArvore noArv, NoArvore noEst, char
            lado, String regra, int i, Stack S) throws IOException
877     {
            if(i>0)
879         {
                if(regra.equals(" l  ") || regra.equals(" r  "))
881             {
                    F = aplicaRegraLandRor(F, noEst);
883                 EscreveNoF(F);
                    //return F;
885             }
                else if (regra.equals(" r    ") || regra.equals(" l    "))
887             {
                    F = aplicaRegraRnotAndLnotOr(F, noEst, noArv);
889                 EscreveNoF(F);
                }
891             else if (regra.equals(" l    ") || regra.equals(" r    "))
                {
893                 F = aplicaRegraLnotnotRnotnot(F, noEst, noArv);
                    EscreveNoF(F);
895             }
                else if(regra.equals(" l  ") || regra.equals(" r  ") || regra.equals("
                    l    ") || regra.equals(" r    "))
897             {
                    F = aplicaRegraLorRand(F, noEst, lado);
899                 EscreveNoF(F);
                }
901             else if (regra.equals(" r  ") || regra.equals(" l  ") || regra.equals("
                    l    ") || regra.equals(" r    "))
                {
903                 F = aplicaRegraRparatodoLexiste(F, noEst);
                    EscreveNoF(F);
905             }
                else if(regra.equals("cut"))
907             {
                    if(!noEst.direita.rotulo.equals(cstr1) && !noEst.direita.rotulo.
                        equals(cstr2) &&
909                     !noEst.direita.rotulo.equals(cstr3) && !noEst.direita.rotulo.
                            equals(cstr4) &&
                        !noEst.direita.rotulo.equals(cstr5) && !noEst.direita.rotulo.
                            equals(cstr6) &&
911                     !noEst.direita.rotulo.equals(cstr7)) //   um conjunto de  n  s
```

```
                {
913                 F = aplicaRegraCorte2(F, noEst.direita, noEst.direita.direita,
                        lado, S);
                    EscreveNoF(F);
915                 //return F;
                }
917             else
                {
919                 System.out.println("ERRO!");
    //                int j = 0;
921 //                while(S != null)
    //                {
923 //                    F.No[j] = (Literal)S.lastElement();
    //                    S.pop();//desempilhe o elemento do topo de S ;
925 //                    j = j +1;
    //                }
927             }
            }
929     }
        if(!noEst.rotulo.equals(cstr1) && !noEst.rotulo.equals(cstr2) &&
931         !noEst.rotulo.equals(cstr3) && !noEst.rotulo.equals(cstr4) &&
            !noEst.rotulo.equals(cstr5) && !noEst.rotulo.equals(cstr6) &&
933         !noEst.rotulo.equals(cstr7)) //   um conjunto de n s
        {
935         return null;
        }
937     else
        {
939         if(i>0)
            {
941             NoArvore noEstAux = noEst;
                noEst = noEst.direita;
943             if(!noEst.rotulo.equals(cstr1) && !noEst.rotulo.equals(cstr2) &&
                    !noEst.rotulo.equals(cstr3) && !noEst.rotulo.equals(cstr4) &&
945                 !noEst.rotulo.equals(cstr5) && !noEst.rotulo.equals(cstr6) &&
                    !noEst.rotulo.equals(cstr7))
947             {
                    noEst = noEstAux.esquerda;
949                 ladoEstrutura = "E";

951             }
            }
953         regra = buscaRegraSequentes(noArv, noEst);
            while(regra.equals(""))
955         {
                noEst = noEst.direita;
957             regra = buscaRegraSequentes(noArv, noEst);
                if(regra.equals(""))
959             {
                    noEst = noEst.direita;
961                 regra = buscaRegraSequentes(noArv, noEst);

963             }
            }
965     }
        if(noEst.tipo.equals("  '"))
967     {
```

```
969                 if(!noEst.direita.rotulo.equals(cstr1) && !noEst.direita.rotulo.
                        equals(cstr2) &&
                            !noEst.direita.rotulo.equals(cstr3) && !noEst.direita.rotulo
                                .equals(cstr4) &&
971                             !noEst.direita.rotulo.equals(cstr5) && !noEst.direita.rotulo
                                .equals(cstr6) &&
                            !noEst.direita.rotulo.equals(cstr7)) //   um conjunto de
                                n  s
973                 {
                        if(!ladoEstrutura.equals("E"))
975                     {
                            F.direita = constroiSequentes(F, noArv, noEst, 'D', regra,
                                1, S);
977                     }
                        else
979                     {
                            F.esquerda = constroiSequentes(F, noArv, noEst, 'E', regra,
                                1, S);
981                     }
                    }
983                 else
                    {
985                     F.esquerda = constroiSequentes(F,noArv,noEst,'E', regra, 1, S);
                        F.direita = constroiSequentes(F, noArv, noEst, 'D', regra, 1, S)
                            ;
987                 }
            }
989         else
            {
991             F.direita = constroiSequentes(F, noArv, noEst,'D', regra, 1, S );
            }
993         if(noEst.tipo.equals("  "))
            {
995             F.esquerda = constroiSequentes(F, noArv, noEst, 'E', regra, 1, S );
            }
997         return F;
        }//Fim constroiSequentes
999
        public static NoF aplicaRegraLandRor(NoF F, NoArvore noEst)
1001    {
            boolean achou = false;
1003        Literal[] auxF = new Literal[F.No.length];// = new Literal[F];
            int i = 0;
1005        int j = 0;
            int idx = buscaIndice(F, noEst.posicao);
1007        int[] par = posicaoParenteses(F,idx);

1009        int tam = F.No.length;

1011        while(i < tam)
            {
1013    //              if((i == par[0]) || (i == par[1]))
        //              {
1015    //                  i = i + 1;
        //              }
1017    //              else
```

```
            if(F.No[i].posicao == noEst.posicao)
1019        {
                auxF[j] = new Literal();
1021            auxF[j].rotulo = ",";
                auxF[j].posicao = noEst.posicao;
1023            j = j + 1;
                i = i + 1;
1025        }
            else
1027        {
                auxF[j] = F.No[i];
1029            j = j + 1;
                i = i + 1;
1031        }
        }
1033    F.No = auxF;
        return F;
1035 }//Fim aplicaRegraLandRor

1037 public static int buscaIndice(NoF F,int posicao)
    {
1039    /* Busca  ndice  de  n  em um array */
        int tam = F.No.length;
1041    int i = 0;
        int idx = 0;
1043    boolean achou = false;

1045    while((i < tam) && (achou == false))
        {
1047        if(F.No[i].posicao == posicao)
            {
1049            achou = true;
                idx = i;
1051        }
            i = i + 1;
1053    }
        return idx;
1055 }//Fim buscaIndice

1057 public static int[] posicaoParenteses(NoF F, int idx)
    {
1059    boolean achou = false;
        int[] auxpar = new int[F.No.length];
1061    int tam = F.No.length;
        int j = 0;
1063    int i = idx - 1;;
        /* Procura o   (   mais pr ximo do n  de  ndice  idx. */
1065    while((i >= 0) && (achou == false))
        {
1067        if(F.No[i].rotulo.equals("("))
            {
1069            achou = true;
                auxpar[j] = i;
1071            j = j + 1;
            }
1073        i = i + 1;
        }
```

```
1075            achou = false;
                i = idx + 1;
1077            while((i < tam) && (achou == false))
                {
1079                if(F.No[i].rotulo.equals(")"))
                    {
1081                    achou = true;
                        auxpar[j] = i;
1083                    j = j + 1;
                    }
1085                i = i + 1;
                }
1087            //Transfere valores de auxpar para par
                int[] par = new int[j];
1089            for(int k = 0; k < j; k++)
                {
1091                par[k] = auxpar[k];
                }
1093
                return par;
1095        }//Fim posicaoParenteses

1097         public static NoF aplicaRegraRnotAndLnotOr(NoF F, NoArvore noEst, NoArvore
                noArv)
            {
1099            boolean achou = false;
                int i = 0;
1101            int j = 0;
                Literal[] auxF = null;
1103            int idx = buscaIndice(F,noEst.posicao);
                NoArvore noNeg;
1105            Stack auxCn1 = buscaCaminho(noArv, noEst.posicao, null);
                NoArvore cn1[] = null;
1107            //Inserindo os valores da Pilha auxCn1 em Cn1
                for(int k = auxCn1.size(); k >= 0; k++)
1109            {
                    cn1[k] = (NoArvore)auxCn1.lastElement();
1111                auxCn1.pop();
                }
1113            noNeg = buscaNoRotulo("  ", cn1);
                int par[] = posicaoParenteses(F,idx);
1115            int tam = F.No.length;
                while(i < tam)
1117            {
                    if(i == par[0] || i == par[1])
1119                {
                        i = i + 1;
1121                }
                    else if(F.No[i].posicao == noEst.posicao)
1123                {
                        auxF[j].rotulo = ",";
1125                    auxF[j].posicao = noEst.posicao;
                        j = j + 1;
1127                    auxF[j].rotulo = noNeg.rotulo;
                        auxF[j].posicao = noNeg.posicao;
1129                    j = j + 1;
                        i = i + 1;
```

```
1131            }
                else
1133            {
                    auxF[j] = F.No[i];
1135                j = j + 1;
                    i = i + 1;
1137            }
            }
1139        F.No = auxF;
            return F;
1141    }//Fim aplicaRegraRnotAndLnotOr

1143    public static Stack buscaCaminho(NoArvore no,int pos, Stack caminho)
        {
1145        Literal noNulo = new Literal();
            noNulo.posicao = -1;
1147
            if(no == null)
1149        {
                caminho.push(noNulo); //empilhe noNulo no topo do caminho;
1151            return caminho;
            }
1153        else
            {
1155            caminho.push(no);//empilhe no:pos no caminho;
                if(no.posicao == pos)
1157            {
                    return caminho;
1159            }
                else
1161            {
                    caminho = buscaCaminho(no.direita, pos, caminho);
1163                if(caminho.lastElement() instanceof Literal)
                    {
1165                    caminho.pop();
                    }
1167                if(((NoArvore)caminho.lastElement()).posicao == -1)
                    {
1169                    caminho.pop();
                        caminho = buscaCaminho(no.esquerda, pos, caminho);
1171                    if(((NoArvore)caminho.lastElement()).posicao == -1)
                        {
1173                        caminho.pop();
                            caminho.pop();//desempilhe os dois primeiros elementos do
                                topo do caminho;
1175                        caminho.push(noNulo);
                            return caminho;
1177                    }
                        else
1179                    {
                            return caminho;
1181                    }
                    }
1183                else
                    {
1185                    return caminho;
                    }
```

```
1187            }
            }
1189    }//Fim BuscaCaminho

1191    public static NoArvore buscaNoRotulo(String rotulo, NoArvore[] caminho)
        {
1193        /* Busca um n   com um dado r tulo no caminho entre o n   raiz e um
            dado n  . Por exemplo, um n   com r tulo igual a    precede um certo
1195        n  . */
            boolean achou = false;
1197        int i = 0;
            int tam = caminho.length;
1199        while((i <= tam - 2) && (achou == false))
            {
1201            if(caminho[i].rotulo.equals(rotulo))
                    achou = true;
1203        }
            return caminho[i];
1205    }//Fim buscaNoRotulo

1207    public static NoF aplicaRegraLnotnotRnotnot(NoF F, NoArvore noEst, NoArvore
            noArv)
        {
1209        boolean achou = false;
            int i = 0;
1211        int j = 0;
            Literal[] auxF = null;
1213        NoArvore noNeg;
            Stack auxCn1 = buscaCaminho(noArv, noEst.posicao, null);
1215        NoArvore cn1[] = null;
            //Inserindo os valores da Pilha auxCn1 em Cn1
1217        for(int k = auxCn1.size(); k > 0; k--)
            {
1219            cn1[k] = (NoArvore)auxCn1.lastElement();
                auxCn1.pop();
1221        }
            noNeg = buscaNoRotulo("  ", cn1);
1223        int tam = F.No.length;
            for(i = 0; i < tam; i++)
1225        {
                if(F.No[i].posicao == noNeg.posicao && F.No[i].posicao == noEst.posicao)
1227            {
                    auxF[j] = F.No[i];
1229                j = j + 1;
                }
1231        }
            F.No = auxF;
1233
            return F;
1235    }//Fim aplicaRegraLnotnotRnotnot

1237    public static NoF aplicaRegraLorRand(NoF F, NoArvore noEst, char lado)
        {
1239        int[] par = null;
            boolean achou = false;
1241        int i = 0;
            int j = 0;
```

```
1243          int idx = buscaIndice(F, noEst.posicao);
              int tam = F.No.length;
1245          par = posicaoParenteses(F,idx);
              Literal[] auxF = null;
1247          if(lado == 'd')
              {
1249              while(i < tam)
                  {
1251                  if(((i >= par[0]) && (i <= idx)) || (i == par[1]))
                      {
1253                      i = i + 1;
                      }
1255                  else
                      {
1257                      auxF[j] = F.No[i];
                          j = j + 1;
1259                      i = i + 1;
                      }
1261              }
              }
1263          if(lado == 'e')
              {
1265              while(i < tam)
                  {
1267                  if(((i >= idx) && (i <= par[1])) || (i == par[0]))
                      {
1269                      i = i + 1;
                      }
1271                  else
                      {
1273                      auxF[j] = F.No[i];
                          j = j + 1;
1275                      i = i + 1;
                      }
1277              }
              }
1279          F.No = auxF;

1281          return F;
          }//Fim aplicaRegraLorRand
1283
          public static NoF aplicaRegraRparatodoLexiste(NoF F, NoArvore noEst)
1285      {
              /* armazena as posi  es de todos os quantificadores que devem ser
1287          reduzidos juntos. */
      //        int posQuants[] = noEstposNRJ[];
1289  //        int tam = posQuants.length;
      //        posQuants[tam] = noEst.posicao;
1291  //        tam = posQuants.length;
      //        int j =0;
1293  //        tam = F.No.length;
      //        int t = idx.length;
1295  //        int i = 0;
      //        Literal[] auxF = null;
1297  //        while(i<tam)
      //        {
1299  //            for(int k = 0; k < t; k++)
```

```
//            {
//                if(F.No[i].posicao == posQuants[k])
//                {
//                    i = i + 2;
//                    /* para    eliminar    a rela  o associada ao quantificador. */
//                }
//                else
//                {
//                    auxF[j] = F.No[i];
//                    j = j + 1;
//                }
//            }
//        }
//        F.No = auxF;
        return F;
    }//Fim aplicaRegraRparatodoLexiste

    public static NoF aplicaRegraCorte(NoF F, NoArvore noEst0, NoArvore noEst1, char lado, Stack S)
    {
        boolean continua = true;
        int cont = 0;
        int idNoAxioma;
        int idNo;
        Stack axioma = new Stack();
        Stack seqCons = new Stack();
        Stack sucedente = new Stack();
        //Stack sucedente = null;
        Stack antecedente = new Stack();
        //Stack antecedente = null;
        //NoF sequente = null;
        Stack sequente = new Stack();
        /* O n  noEst  um array com os 2 n s que formaram a conex o. O n com menor posi  o , faz parte do axioma inicial. */
        if(noEst0.posicao < noEst1.posicao)
        {
            idNoAxioma = noEst0.posicao;
            idNo = noEst1.posicao;
        }
        else
        {
            idNoAxioma = noEst1.posicao;
            idNo = noEst0.posicao;
        }
        if(lado == 'D')
        {
//            /* temos o axioma inicial para o ramo direito do sequentes */
//            axioma = buscaSequenteAxiomaD(F, idNoAxioma);
//            /* temos o consequente do sequente a ser provado */
//            sucedente = buscaAntecedenteSequente(axioma, idNoAxioma);
//            /* temos um sequente do consequente da f rmula. Falta extrair seu antecedente. */
//            seqCons = buscaSequenteAxiomaD(F, idNo);
//            /* temos o antecedente do sequente a ser provado */
//            antecedente = buscaAntecedenteSequente(seqCons, idNoAxioma);
        }
```

```
        else
1355    {
            /* temos o axioma inicial para o ramo esquerdo do sequentes */
1357        axioma = buscaSequenteAxiomaE(F, idNoAxioma);
            /* temos o antecedente do sequente a ser provado */
1359        antecedente = buscaConsequenteSequente(axioma);
            /* temos um sequente do consequente da f rmula. Falta extrair seu
                sucedente. */
1361        seqCons = buscaSequenteAxiomaE(F, idNo);
            /* temos o sucedente do sequente a ser provado */
1363        sucedente = buscaConsequenteSequente(seqCons);
            /* O sequente a ser provado    a jun   o: antecedente +     + sucedente
                */
1365        /* S recebe seus elementos na ordem da direita para esquerda. */
        }
1367
        int k = 0;
1369    while(sucedente.empty() != true)//while(sucedente != null)
        {
1371        S.push(sucedente.lastElement());//empilhe o elemento do topo de
                sucedente em S ;
            sucedente.pop();//sucedente.No[sucedente.No.length - k] = null;//
                sucedente.pop();//desempilhe o elemento do topo de sucedente;
1373        S.push("   ");//empilhe '   ' em S;
            k = k + 1;
1375    }
        int t = 0;
1377    while(antecedente.empty() != true)//while(antecedente != null)
        {
1379        S.push(antecedente.lastElement());//S.push(antecedente.No[antecedente.No
                .length - t]);//empilhe o elemento do topo de antecedente em S ;
            antecedente.pop();//antecedente.No[antecedente.No.length - k] = null;//
                antecedente.pop();//desempilhe o elemento do topo de antecedente;
1381    }
        F.No = new Literal[axioma.size()];
1383    int cont2 = 0;
        while(axioma.empty() == false){
1385        //F.No[cont2] = new Literal();
            F.No[cont2] = (Literal)axioma.lastElement();
1387        axioma.pop();
            cont2++;
1389    }

1391    return F;
    }//Fim aplicaRegraCorte
1393
    public static NoF aplicaRegraCorte2(NoF F, NoArvore noEst0, NoArvore noEst1,
        char lado, Stack S)
1395{
        boolean continua = true;
1397    int cont = 0;
        int idNoAxioma;
1399    int idNo;
        //Stack axioma = new Stack();
1401    //Stack seqCons = new Stack();

1403    Stack antecedente1 = new Stack();
```

```
        Stack sucedente1 = new Stack();
1405    Stack antecedente2 = new Stack();
        Stack sucedente2 = new Stack();
1407
        //Stack sequente = new Stack();
1409    /* O n   noEst    um array com os 2 n s que formaram a conex o. O n
        com menor posi   o , faz parte do axioma inicial. */
1411    if(noEst0.posicao < noEst1.posicao)
        {
1413        idNoAxioma = noEst0.posicao;
            idNo = noEst1.posicao;
1415    }
        else
1417    {
            idNoAxioma = noEst1.posicao;
1419        idNo = noEst0.posicao;
        }
1421    if(lado == 'D')
        {
1423        //Pega primeiro axioma e verifica antecedente e sucedente
            int par[] = new int[3];
1425        boolean achou1 = false;
            boolean achou2 = false;
1427        int contachou1 = 0;
            int contachou2 = 0;
1429        NoF auxF = F;
            for(int i = 0; i < F.No.length; i++){
1431            if(F.No != null){
                    if(F.No[i].posicao == idNoAxioma){
1433                    while(achou1 == false){
                            i++;
1435                        if(F.No[i].rotulo.equals("(")){
                                contachou1++;
1437                        }
                            if(F.No[i].rotulo.equals(")") && contachou1 > 0){
1439                            contachou1--;
                            }
1441                        if(F.No[i].rotulo.equals(")") && contachou1 == 0){
                                achou1 = true;
1443                            par[1] = i;
                            }
1445                    }
                        while(achou2 == false){
1447                        i--;
                            if(F.No[i].rotulo.equals(")")){
1449                            contachou2++;
                            }
1451                        if(F.No[i].rotulo.equals("(") && contachou2 > 0){
                                contachou2--;
1453                        }
                            if(F.No[i].rotulo.equals("(") && contachou2 == 0){
1455                            achou2 = true;
                                par[0] = i;
1457                        }
                    }
1459                }
                    else{
```

```
1461                        continue;
                        }
1463                }
                    else{
1465                    break;
                    }
1467            }
                for(int i = par[1]; i > par[0]; i--){
1469                if(F.No[i].rotulo.equals(cstr1) || F.No[i].rotulo.equals(cstr2) ||
                        F.No[i].rotulo.equals(cstr3) || F.No[i].rotulo.equals(cstr4) ||
1471                    F.No[i].rotulo.equals(cstr5) || F.No[i].rotulo.equals(cstr6) ||
                        F.No[i].rotulo.equals(cstr7)){
1473                    par[2] = i;
                        break;
1475                }
                }
1477            for(int i = par[0]; i < par[1]; i++){
                    if(i < par[2])
1479                    antecedente1.push(F.No[i]);
                    if(i == par[2])
1481                    continue;
                    if(i > par[2])
1483                    sucedente1.push(F.No[i]);
                }
1485            //Fim Pega primeiro axioma e verifica antecedente e sucedente
                //Pega Segundo axioma e verifica antecedente e sucedente
1487            achou1 = false;
                achou2 = false;
1489            contachou2 = 0;
                for(int i = 0; i < F.No.length; i++){
1491                if(F.No != null){
                        if(F.No[i].posicao == idNo){
1493                        while(achou1 == false){
                                i++;
1495                            if(F.No[i].rotulo.equals("(")){
                                    contachou1++;
1497                            }
                                if(F.No[i].rotulo.equals(")") && contachou1 > 0){
1499                                contachou1--;
                                }
1501                            if(F.No[i].rotulo.equals(")") && contachou1 == 0){
                                    achou1 = true;
1503                                par[1] = i;
                                }
1505                        }
                            while(achou2 == false){
1507                            i--;
                                if(F.No[i].rotulo.equals(")")){
1509                                contachou2++;
                                }
1511                            if(F.No[i].rotulo.equals("(") && contachou2 > 0){
                                    contachou2--;
1513                            }
                                if(F.No[i].rotulo.equals("(") && contachou2 == 0){
1515                                achou2 = true;
                                    par[0] = i;
1517                            }
```

```
                        }
1519                    }
                        else{
1521                        continue;
                        }
1523                }
                    else{
1525                    break;
                    }
1527            }
                for(int i = par[1]; i > par[0]; i--){
1529                if(F.No[i].rotulo.equals(cstr1) || F.No[i].rotulo.equals(cstr2) ||
                        F.No[i].rotulo.equals(cstr3) || F.No[i].rotulo.equals(cstr4) ||
1531                    F.No[i].rotulo.equals(cstr5) || F.No[i].rotulo.equals(cstr6) ||
                        F.No[i].rotulo.equals(cstr7)){
1533                    par[2] = i;
                        break;
1535                }
                }
1537            for(int i = par[0]; i <= par[1]; i++){
                    if(i < par[2])
1539                    antecedente2.push(F.No[i]);
                    if(i == par[2])
1541                    continue;
                    if(i > par[2])
1543                    sucedente2.push(F.No[i]);
                }
1545            //Fim Pega Segundo axioma e verifica antecedente e sucedente
                //Altera o valor de F
1547            int tamF = 0;
                for(int i = 0; i < F.No.length; i++){
1549                if(i < antecedente2.size()){
                        F.No[i] = (Literal)antecedente2.get(i);
1551                    tamF++;
                    }
1553                if(i == antecedente2.size()){
                        F.No[i].rotulo = cstr1;
1555                    tamF++;
                    }
1557                if(i > antecedente2.size()){
                        F.No[i] = null;
1559                }
                }
1561            int contPilha = 0;
                for(int i = tamF; i < F.No.length; i++){
1563                if(contPilha < antecedente1.size()){
                        F.No[i] = (Literal)antecedente1.get(contPilha);
1565                    contPilha++;
                        continue;
1567                }
                    if(contPilha == antecedente1.size()){
1569                    continue;
                    }
1571                if(contPilha > antecedente1.size()){
                        F.No[i] = null;
1573                }
                }
```

```
1575        }
            else
1577        {
                //Pega primeiro axioma e verifica antecedente e sucedente
1579            int par[] = new int[3];
                boolean achou1 = false;
1581            boolean achou2 = false;
                int contachou1 = 0;
1583            int contachou2 = 0;
                for(int i = 0; i < F.No.length; i++){
1585                if(F.No[i] != null){
                        if(F.No[i].posicao == idNoAxioma){
1587                        while(achou1 == false){
                                i++;
1589                            if(F.No[i].rotulo.equals("(")){
                                    contachou1++;
1591                            }
                                if(F.No[i].rotulo.equals(")") && contachou1 > 0){
1593                                contachou1--;
                                }
1595                            if(F.No[i].rotulo.equals(")") && contachou1 == 0){
                                    achou1 = true;
1597                                par[1] = i;
                                }
1599                        }
                            while(achou2 == false){
1601                            i--;
                                if(F.No[i].rotulo.equals(")")){
1603                                contachou2++;
                                }
1605                            if(F.No[i].rotulo.equals("(") && contachou2 > 0){
                                    contachou2--;
1607                            }
                                if(F.No[i].rotulo.equals("(") && contachou2 == 0){
1609                                achou2 = true;
                                    par[0] = i;
1611                            }
                            }
1613                    }
                        else{
1615                        continue;
                        }
1617                }
                    else{
1619                    break;
                    }
1621            }
                for(int i = par[1]; i > par[0]; i--){
1623                if(F.No[i].rotulo.equals(cstr1) || F.No[i].rotulo.equals(cstr2) ||
                        F.No[i].rotulo.equals(cstr3) || F.No[i].rotulo.equals(cstr4) ||
1625                    F.No[i].rotulo.equals(cstr5) || F.No[i].rotulo.equals(cstr6) ||
                        F.No[i].rotulo.equals(cstr7)){
1627                    par[2] = i;
                        break;
1629                }
                }
1631            for(int i = par[0]; i < par[1]; i++){
```

```
                if(i < par[2])
1633                antecedente1.push(F.No[i]);
                if(i == par[2])
1635                continue;
                if(i > par[2])
1637                sucedente1.push(F.No[i]);
            }
1639        //Fim Pega primeiro axioma e verifica antecedente e sucedente
            //Pega Segundo axioma e verifica antecedente e sucedente
1641        achou1 = false;
            achou2 = false;
1643        contachou2 = 0;
            for(int i = 0; i < F.No.length; i++){
1645            if(F.No != null){
                    if(F.No[i].posicao == idNo){
1647                    while(achou1 == false){
                            i++;
1649                        if(F.No[i].rotulo.equals("(")){
                                contachou1++;
1651                        }
                            if(F.No[i].rotulo.equals(")") && contachou1 > 0){
1653                            contachou1--;
                            }
1655                        if(F.No[i].rotulo.equals(")") && contachou1 == 0){
                                achou1 = true;
1657                            par[1] = i;
                            }
1659                    }
                        while(achou2 == false){
1661                        i--;
                            if(F.No[i].rotulo.equals(")")){
1663                            contachou2++;
                            }
1665                        if(F.No[i].rotulo.equals("(") && contachou2 > 0){
                                contachou2--;
1667                        }
                            if(F.No[i].rotulo.equals("(") && contachou2 == 0){
1669                            achou2 = true;
                                par[0] = i;
1671                        }
                        }
1673                }
                    else{
1675                    continue;
                    }
1677            }
                else{
1679                break;
                }
1681        }
            for(int i = par[1]; i > par[0]; i--){
1683            if(F.No[i].rotulo.equals(cstr1) || F.No[i].rotulo.equals(cstr2) ||
                    F.No[i].rotulo.equals(cstr3) || F.No[i].rotulo.equals(cstr4) ||
1685                F.No[i].rotulo.equals(cstr5) || F.No[i].rotulo.equals(cstr6) ||
                    F.No[i].rotulo.equals(cstr7)){
1687                par[2] = i;
                    break;
```

```
1689                }
            }
1691            for(int i = par[0]; i <= par[1]; i++){
                if(i < par[2])
1693                antecedente2.push(F.No[i]);
                if(i == par[2])
1695                continue;
                if(i > par[2])
1697                sucedente2.push(F.No[i]);
            }
1699            //Fim Pega Segundo axioma e verifica antecedente e sucedente
            //Altera o valor de F
1701            int tamF = 0;
            for(int i = 0; i < F.No.length; i++){
1703                if(i < antecedente1.size()){
                    F.No[i] = (Literal)antecedente1.get(i);
1705                tamF++;
                }
1707                if(i == antecedente1.size()){
                    F.No[i].rotulo = cstr1;
1709                tamF++;
                }
1711                if(i > antecedente1.size()){
                    F.No[i] = null;
1713                }
            }
1715            int contPilha = 0;
            for(int i = tamF; i < F.No.length; i++){
1717                if(contPilha < sucedente2.size()){
                    F.No[i] = (Literal)sucedente2.get(contPilha);
1719                contPilha++;
                    continue;
1721                }
                if(contPilha == sucedente2.size()){
1723                continue;
                }
1725                if(contPilha > sucedente2.size()){
                    F.No[i] = null;
1727                }
            }
1729        }

1731

        return F;
1733    }//Fim aplicaRegraCorte

1735    public static Stack buscaSequenteAxiomaD(NoF F, int idNo)
    {
1737        /* pega o  ndice  de   )   na f rmula que delimita o axioma inicial ou
        sequente que estamos procurando. Essa busca   feita da direita
1739        para a esquerda. */
        boolean continua = true;
1741        int cont = 0;
        int idx = buscaIndice(F,idNo);
1743        Stack auxF = new Stack();
        Stack axioma = new Stack();
1745
```

```
            while(continua == true)
1747        {
                idx = idx + 1;
1749            if(F.No[idx] != null | F.No[idx].rotulo != null)
                {
1751                if(F.No[idx].rotulo.equals(")"))
                    {
1753                    cont = cont + 1;
                        continua = false;
1755                }
                    else
1757                {
                        continua = false;
1759                }
                }
1761        }
            idx = idx - 1;
1763        continua = true;
            cont = 0;
1765        /* empilha em auxF o axioma inicial */
            while(continua == true)
1767        {
                if(F.No[idx] != null)
1769            {
                    if(F.No[idx].rotulo.equals(")"))
1771                {
                        cont = cont + 1;
1773                }
                    else if(F.No[idx].rotulo.equals("("))
1775                {
                        cont = cont - 1;
1777                    if(cont == 0)
                        {
1779                        continua = false;
                        }
1781                }
                    auxF.push(F.No[idx]); //empilhe em F:No[idx] em auxF;
1783                idx = idx - 1 ;
                }
1785        }
            /* o axioma agora    emplilhado em axioma, sentido esquerda - direita*/
1787        while(auxF.empty() != true)
            {
1789            axioma.push(auxF.lastElement());//axioma = (NoF)auxF.lastElement();//
                    empilhe o elemento do topo de auxF em axioma;
                auxF.pop();//desempilhe o elemento do topo de auxF;
1791        }
            return axioma;
1793    }//Fim buscaSequenteAxiomaD

1795    public static Stack buscaAntecedenteSequente(Stack sequente, int idNo)
        {
1797        /* Armazena o antecedente de um Sequente/axioma. Para isso,
            preciso desempilhar os elementos at   o     e contar os    )    */
1799        //NoF sequente
            //NoF temp;
1801        //NoF auxSeq;
```

```
            //NoF antecedente
1803        boolean continua = true;
            int cont = 0;
1805        Stack temp = new Stack();
            Stack auxSeq = new Stack();
1807        Stack antecedente = new Stack();
            //NoF axioma = null;
1809        cont = 0;

1811        //Inserindo os valores da Pilha auxCn1 em Cn1
    //        Stack sequente = null;
1813 //        for(int k = 0; k < auxSequente.No.length; k++)
    //        {
1815 //            sequente.push(auxSequente.No[k]);
    //        }
1817        while(continua == true)
            {
1819            if(((Literal)sequente.lastElement()).posicao == idNo)//atributo posicao
                    do elemento do topo de sequente = idNo ent o
                {
1821                temp.push(sequente.lastElement());//empilhe o elemento do topo de
                        sequente em temp;
                    sequente.pop();//desempilhe o elemento do topo de sequente;
1823                continua = false;
                }
1825            temp.push(sequente.lastElement());//empilhe o elemento do topo de
                    sequente em temp;
                sequente.pop();//desempilhe o elemento do topo de sequente;
1827        }
            /* Retira os par nteses excedentes */
1829        cont = 0;
            continua = true;
1831 //        while(continua == true)
    //        {
1833 //            if(((Literal)sequente.lastElement()).rotulo.equals(")"))
    //            {
1835 //                cont = cont + 1;
    //                auxSeq.push(sequente.lastElement());
1837 //                sequente.pop();
    //            }
1839 //            else if(((Literal)sequente.lastElement()).rotulo.equals("("))
    //            {
1841 //                if(cont > 1)
    //                {
1843 //                    cont = cont - 1;
    //                    auxSeq.push(sequente.lastElement());
1845 //                    sequente.pop();
    //                }
1847 //                else if(cont == 0)
    //                {
1849 //                    continua = false;
    //                }
1851 //            }
    //        }
1853 //        while(auxSeq.empty() != true)
    //        {
1855 //            antecedente.push(auxSeq.lastElement());//antecedente = (NoF)auxSeq.
```

```
        lastElement();
//            auxSeq.pop();
//        }
        return sequente;
    }//Fim buscaAntecedenteSequente

    public static Stack buscaSequenteAxiomaE(NoF F, int idNo)
    {
        /* pega o  ndice  de   (   na f rmula que delimita o axioma inicial ou
        sequente que estamos procurando. Essa busca   feita da esquerda
        para direita. */
        boolean continua = true;
        int cont = 0;
        Stack auxF = new Stack();
        Stack axioma = new Stack();
        int idx = buscaIndice(F,idNo);


        while(continua == true)
        {
            idx = idx - 1;
            if(F.No != null && idx < F.No.length)
            {
                if(F.No[idx].rotulo.equals("("))
                {
                    cont = cont + 1;
                }
                else
                {
                    continua = false;
                }
            }
        }
        idx = idx + 1;
        continua = true;
        cont = 0;
        /* empilha em auxF o axioma inicial */
        while(continua == true)
        {
            if(F.No[idx].rotulo.equals("("))
            {
                cont = cont + 1;
            }
            else if(F.No[idx].rotulo.equals(")"))
            {
                cont = cont - 1;
                if(cont == 0)
                {
                    continua = false;
                }
            }
            auxF.push(F.No[idx]);
            idx = idx + 1 ;
        }
        /* o axioma recebe auxF, que j  est  no sentido esquerda - direita
        ( ltimo  elemento da direita = elemento do topo da pilha) */

```

```
            axioma = auxF;//axioma.push(auxF.lastElement());//axioma = (NoF)auxF.
                lastElement();
1913
            return axioma;
1915        }//Fim buscaSequenteAxiomaE

1917        public static Stack buscaConsequenteSequente(Stack sequente)
            {
1919            /* Armazena o consequente de um Sequente/axioma. Para isso,
                preciso desempilhar os elementos ap s o     e contar os    (    */
1921            boolean continua = true;
                int cont = 0;
1923            Stack consequente = new Stack();
                Stack auxSequente = new Stack();
1925            Stack auxSequente2 = new Stack();
                //auxSequente = sequente;
1927            /* desempilha todos os elementos que antecedem     */
                while(!((Literal)sequente.lastElement()).rotulo.equals("   "))
1929            {
                    auxSequente2.push(sequente.lastElement());
1931                sequente.pop();
                }
1933            /* desempilha o     */
                sequente.pop();
1935    //        while(auxSequente2.isEmpty() == false){
        //            auxSequente.push(auxSequente2.lastElement());
1937    //            auxSequente2.pop();
        //        }
1939            /* Descobre o consequente,     eliminando     os par nteses excedentes */
                cont = 0;
1941            while(continua == true)
                {
1943                if(((Literal)auxSequente2.lastElement()).rotulo.equals("("))
                    {
1945                    cont = cont + 1;
                        consequente.push(auxSequente2.lastElement());//consequente = (NoF)
                            sequente.lastElement();
1947                    auxSequente2.pop();
                    }
1949                else if(((Literal)auxSequente2.lastElement()).rotulo.equals(")"))
                    {
1951                    if(cont >= 1)
                        {
1953                        cont = cont - 1;
                            consequente.push(auxSequente2.lastElement());//consequente = (
                                NoF)sequente.lastElement();
1955                        auxSequente2.pop();
                        }
1957                    else if(cont == 0)
                        {
1959                        continua = false;
                        }
1961                }
                    else
1963                {
                        /* o atributo rotulo do elemento do topo de sequente    um
1965                    literal */
```

```
                    consequente.push(auxSequente2.lastElement());//consequente = (
                        NoF)sequente.lastElement();
1967                    auxSequente2.pop();
                }
1969        }
            consequente = auxSequente2;
1971        return consequente;
        }//Fim buscaConsequenteSequente
1973
        public static String buscaRegraSequentes(NoArvore noArv, NoArvore noEst)
1975    {
            String regra;
1977        boolean resp;
            Stack caminho = new Stack();
1979        Stack cn1 = null;

1981        cn1 = buscaCaminho(noArv, noEst.posicao, caminho);
            resp = constaNoRotulo("  ", cn1);
1983        /* Se a nega  o precede o n  , busca a regra em uma tabela X, que deve
            representar a tabela 8, sen o , na tabela Y, representando a
1985        tabela 7. */
            if(resp == true)
1987            regra = buscaRegra(noEst, "X");//Tabela 8
            else
1989            regra = buscaRegra(noEst, "Y");//Tabela 7

1991        return regra;
        }//Fim buscaRegraSequentes
1993
        public static boolean constaNoRotulo(String rotulo, Stack auxcaminho)
1995    {
            /* Checa um n  com um dado r tulo est  no caminho entre o n  raiz e
1997        um dado n  . Por exemplo, um n  com r tulo igual a  precede um
            certo n  . */
1999        boolean achou = false;
            int i = 0;
2001        int tam = auxcaminho.size();//tamanho do caminho;
            NoArvore caminho[] = new NoArvore[auxcaminho.size()];
2003        for(int k = auxcaminho.size(); k > 0; k--)
            {
2005            caminho[k-1] = (NoArvore)auxcaminho.lastElement();
                auxcaminho.pop();
2007        }
            while((i <= tam - 2) && (achou == false))
2009        {
                if(caminho[i].rotulo.equals(rotulo))
2011            {
                    achou = true;
2013            }
                i++;
2015        }
            achou = false;
2017        return achou;
        }//Fim buscaRegraSequentes
2019
        public static String buscaRegra(NoArvore NoEst, String tabela)
2021    {
```

```
        String regra = "";
2023
        if(tabela.equals("X"))
2025    {
            if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("  ") && NoEst.
                polaridade == 1)
2027        { regra = " r    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("  ") && NoEst.
                polaridade == 0)
2029        { regra = " l    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 1)
2031        { regra = " r    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
2033        { regra = " l    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
2035        { regra = " l    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 1)
2037        { regra = " r    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
2039        { regra = " l    "; return regra; }
            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 1)
2041        { regra = " r    "; return regra; }
            else
2043        { return regra; }
        }
2045    else if (tabela.equals("Y"))
        {
2047        if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 1)
            { regra = " l  "; return regra; }
2049        else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
            { regra = " r  "; return regra; }
2051        else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("  ") && NoEst.
                polaridade == 1)
            { regra = ""; return regra; }
2053        else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("  ") && NoEst.
                polaridade == 0)
            { regra = ""; return regra; }
2055        else if(NoEst.tipo.equals("  '") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
            { regra = ""; return regra; }
2057        else if(NoEst.tipo.equals("  '") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
            { regra = ""; return regra; }
2059        else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 0)
            { regra = " r  "; return regra; }
2061        else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                polaridade == 1)
            { regra = " l  "; return regra; }
```

```
2063            else if(NoEst.tipo.equals("  '") && NoEst.rotulo.equals("   ") && NoEst.
                    polaridade == 1)
                { regra = "cut"; return regra; }
2065            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                    polaridade == 0)
                { regra = " r  "; return regra; }
2067            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                    polaridade == 1)
                { regra = " l  "; return regra; }
2069            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                    polaridade == 1)
                { regra = ""; return regra; }
2071            else if(NoEst.tipo.equals("  ") && NoEst.rotulo.equals("   ") && NoEst.
                    polaridade == 0)
                { regra = ""; return regra; }
2073            else
                {
2075                return regra;
                }
2077        }
            else
2079        { return regra;}

2081

2083    }//Fim buscaRegra

2085    public static String imprimirArvoreProva(NoArvore noEst, boolean NovoArq) throws
            IOException, NoSuchElementException
        {
2087        if(NovoArq == true)
            {
2089            imgFile.createNewFile();
            }
2091
            int i = 0;
2093
            String v1 = noEst.rotulo +
2095                        " " +
                            noEst.polaridade +
2097                        "|" +
                            noEst.tipo +
2099                        "|" +
                            noEst.posicao;
2101        g1.addVertex(v1);
            if(noEst.direita != null)
2103        {
                String vertice1 = imprimirArvoreProva(noEst.direita, false);
2105            if(vertice1 != null && vertice1 != "" && vertice1 != "null")
                    g1.addEdge(v1, vertice1, new EdgeConv(""));
2107        }
            if(noEst.esquerda != null)
2109        {
                String vertice2 = imprimirArvoreProva(noEst.esquerda, false);
2111            if(vertice2 != null && vertice2 != "" && vertice2 != "null")
                    g1.addEdge(v1, vertice2, new EdgeConv(""));
2113        }
```

```
2115        if(NovoArq == true)
            {
2117            JGraphXAdapter<String, EdgeConv> graphAdapter = new JGraphXAdapter<
                    String, EdgeConv>(g1);
                mxHierarchicalLayout layout = new mxHierarchicalLayout(graphAdapter);
2119            layout.execute(graphAdapter.getDefaultParent());
                BufferedImage image2 = mxCellRenderer.createBufferedImage(graphAdapter,
                    null, 2, Color.WHITE, true, null);
2121            ImageIO.write(image2, "PNG", imgFile);
            }
2123        return v1;

2125    }//Fim imprimirArvoreProva

2127    public static String imprimirArvoreReducao(NoArvore noEst, boolean NovoArq)
            throws IOException
        {
2129        if(NovoArq == true)
            {
2131            imgFileEstParcial.createNewFile();
            }
2133
            String v1 = noEst.rotulo +
2135                        " " +
                            noEst.polaridade +
2137                        "|" +
                            noEst.tipo +
2139                        "|" +
                            noEst.posicao;
2141        g2.addVertex(v1);
            if(noEst.direita != null)
2143        {
                String vertice1 = imprimirArvoreReducao(noEst.direita, false);
2145            if(vertice1 != null && vertice1 != "" && vertice1 != "null")
                    g2.addEdge(v1, vertice1, new EdgeConv(""));
2147        }
            if(noEst.esquerda != null)
2149        {
                String vertice2 = imprimirArvoreReducao(noEst.esquerda, false);
2151            if(vertice2 != null && vertice2 != "" && vertice2 != "null")
                    g2.addEdge(v1, vertice2, new EdgeConv(""));
2153        }

2155        if(NovoArq == true)
            {
2157            JGraphXAdapter<String, EdgeConv> graphAdapter = new JGraphXAdapter<
                    String, EdgeConv>(g2);
                mxHierarchicalLayout layout = new mxHierarchicalLayout(graphAdapter);
2159            layout.execute(graphAdapter.getDefaultParent());
                BufferedImage image2 = mxCellRenderer.createBufferedImage(graphAdapter,
                    null, 2, Color.WHITE, true, null);
2161            ImageIO.write(image2, "PNG", imgFileEstParcial);
            }
2163        return v1;

2165
```

```
    }//Fim imprimirArvoreReducao
2167
    public static void openImage(String caminho)
2169 {
        //janela do programa
2171    JFrame frame = new JFrame("Carregar Imagem");
        //container onde ser o adicionados todos componentes
2173    Container container = frame.getContentPane();

2175    //carrega a imagem passando o nome da mesma
        ImageIcon img = new ImageIcon(caminho);
2177
        //pega a altura e largura
2179    int altura = img.getIconHeight();
        int largura = img.getIconWidth();
2181
        //adiciona a imagem em um label
2183    JLabel label = new JLabel(img);
        //adiciona a altura e largura em outro label
2185    JLabel label2 = new JLabel("Altura: "+altura+"      Largura: "+largura);

2187    //cria o JPanel para adicionar os labels
        JPanel panel = new JPanel();
2189    panel.add(label, BorderLayout.NORTH);
        //panel.add(label2, BorderLayout.SOUTH);
2191
        //adiciona o panel no container
2193    container.add(panel, BorderLayout.CENTER);

2195    frame.pack();
        frame.setVisible(true);
2197
    }//Fim openImage
2199
}//Fim ConverteSequentes
```

ANEXO A – EQUIVALÊNCIAS E MAPEAMENTOS LÓGICOS

- Equivalências lógicas

Figura 26 – Equivalências Lógicas

Nome	Equivalência
Idempotência	$A \wedge A \Leftrightarrow A$
	$A \vee A \Leftrightarrow A$
Comutativa	$A \wedge B \Leftrightarrow B \wedge A$
	$A \vee B \Leftrightarrow B \vee A$
Associativa	$A \wedge (B \wedge C) \Leftrightarrow (A \wedge B) \wedge C$
	$A \vee (B \vee C) \Leftrightarrow (A \vee B) \vee C$
Distributiva	$A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$
	$A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$
Complemento	$A \wedge (B \vee \neg B) \Leftrightarrow A$
	$A \vee (B \wedge \neg B) \Leftrightarrow A$
Absorção	$A \wedge (A \vee B) \Leftrightarrow A$
	$A \vee (A \wedge B) \Leftrightarrow A$
Leis de De Morgan	$\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
	$\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$
Lei da dupla negação	$\neg\neg A \Leftrightarrow A$
Implicação	$A \rightarrow B \Leftrightarrow \neg A \vee B$
	$\neg(A \rightarrow B) \Leftrightarrow A \wedge \neg B$
Equivalência	$A \leftrightarrow B \Leftrightarrow (A \rightarrow B) \wedge (B \rightarrow A)$
	$A \leftrightarrow B \Leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B)$
Contraposição	$A \rightarrow B \Leftrightarrow \neg B \rightarrow \neg A$
Contradição	$A \wedge \neg A \Leftrightarrow \bot$
Tautologia	$A \vee \neg A \Leftrightarrow \top$

Fonte: (PALMEIRA, 2017)

- Mapeamento de Lógica de Descrições para Lógica de Primeira Ordem

Figura 27 – Lógica de Descrições e seu Mapeamento para Lógica de Primeira Ordem

Mapeamento de Conceitos para Lógica de Primeira Ordem[1]					
$\pi_x(A)$	=	$A(x)$	$\pi_y(A)$	=	$A(y)$
$\pi_x(C \sqcap D)$	=	$\pi_x(C) \wedge \pi_x(D)$	$\pi_y(C \sqcap D)$	=	$\pi_y(C) \wedge \pi_y(D)$
$\pi_x(C \sqcup D)$	=	$\pi_x(C) \vee \pi_x(D)$	$\pi_y(C \sqcup D)$	=	$\pi_y(C) \vee \pi_y(D)$
$\pi_x(\exists r.C)$	=	$\exists y.r(x,y) \wedge \pi_y(C)$	$\pi_y(\exists r.C)$	=	$\exists x.r(y,x) \wedge \pi_x(C)$
$\pi_x(\forall r.C)$	=	$\forall y.r(x,y) \rightarrow \pi_y(C)$	$\pi_y(\forall r.C)$	=	$\forall x.r(y,x) \rightarrow \pi_x(C)$
Mapeamento de um TBox $\mathcal{T}$ e um ABox A[2]					
$\pi(\mathcal{T}) =$		$\bigwedge_{C \sqsubseteq D \in \mathcal{T}} \forall x.(\pi_x(C) \rightarrow \pi_x(D))$			
$\pi(A) =$		$\bigwedge_{a:C \in A} \pi_x(C)[x/a] \wedge \bigwedge_{(a,b):r \in A} r(a,b)$			

Fonte: (BAADER et al., 2003)

ANEXO B – ALGORITMOS

• Os algoritmos aqui apresentados foram nomeados e numerados por Palmeira (2017).

Algoritmo 1: Converte Fórmula $\mathcal{ALC}$ para forma Infixa

```
1  Função converteEmInFixa(F[])
2      Fin[];
3      quant;
4      i := 0;
5      para cada elemento m de F[] faça
6          se m.rotulo ≠ ∃ e m.rotulo ≠ ∀ e m.rotulo ≠ '.' então
7              Fin[i] := m;
8              i:=i+1;
9          senão se m.rotulo = ∃ ou m.rotulo = ∀ então
10             quant := m;
11         senão se m.rotulo = '.' então
12             Fin[i] := quant ;
13             i:=i+1;
14     retorna Fin;
```

Complexidade do algoritmo 1: O(n), pois na linha 5 o algoritmo processa n elementos de entrada. As operações dentro do loop aumentam a complexidade em um fator constante, não modificando a complexidade assintótica.

Algoritmo 2: Converte Fórmula na forma Infixa para a forma Pós-fixa

```
1  Função converteEmPosFixa(Fin[])
2      Pilha pilha; construtor := {⊨, ⊑, ⊓, ⊔, ¬, ∃, ∀};
3      para cada elemento m de Fin[] faça
4          se m.rotulo = '(' então
5              empilhe m no topo da pilha;
6          senão se m.rotulo ∉ construtor e m.rotulo ≠ ')' então
7              empilhe m em Fp[];
8          senão se m.rotulo ∈ construtor então
9              empilhe m no topo da pilha;
10         senão se m.rotulo = ')' então
11             enquanto rotulo do elemento do topo da pilha ≠ '(' faça
12                 se o rotulo do elemento topo da pilha ∈ construtor então
13                     empilhe o elemento do topo da pilha em Fp[];
14             desempilhe o elemento do topo da pilha;
15     retorna Fp;
```

Complexidade do algoritmo 2: O(n2), dado que existem dois loops aninhados, linha 4 e linha 12, e o tamanho máximo da pilha será o número de símbolos da fórmula.

Algoritmo 3: CONSTRÓI ÁRVORE DE FÓRMULA

```
1  Função constroiArvore(inStr, inEnd, Fp[], pol, arcos[], posBetal, posGDelta)
2      NoArvore no;
3      trl[];
4      int posBl := posBetal;
5      int posGD := posGDelta;
6      construtor := {⊨, ⊑, ⊓, ⊔, ¬, ∃, ∀};
7      se inStrt > inEnd então
8          retorna null;
9      se Fp[index].rotulo ∈ construtor então
10         trl := buscaTipoPol(Fp[index].rotulo,pol);
11         se trl[0] = β então
12             arcos[0]:= arcos[0] +1;
13             arcos[1]:= arcos[0] +1;
14             no := (Fp[index].rotulo, Fp[index].posicao, pol,trl[0], arcos[0],arcos[1]);
15         senão se trl[0] = β' então
16             arcos[2]:= arcos[2] +1;
17             arcos[3]:= arcos[2] +1;
18             posBl:= Fp[index].posicao;
19             no := (Fp[index].rotulo, Fp[index].posicao, pol,trl[0], arcos[2],arcos[3]);
20         senão se trl[0] = γ ou δ então
21             posGD:= Fp[index].posicao;
22             no := (Fp[index].rotulo, Fp[index].posicao, pol,trl[0]);
23         senão
24             no := (Fp[index].rotulo, pos[index].posicao, pol,trl[0]);
25     senão
26         no := (Fp[index].rotulo, Fp[index].posicao, pol, posBl, posGD);
27     int m := pos[index];
28     index := index - 1;
29     se inStrt = inEnd então
30         retorna no;
31     no.direita := constroiArvore(m+1, inEnd, Fp[], trl[1], arcos, posBl, posGD);
32     no.esquerda := constroiArvore(inStr, m-1, Fp[], trl[2], arcos, posBl, posGD);
33     retorna no;
```

Complexidade do algoritmo 3: O(n2). A cada iteração, no pior caso, reduz-se a fórmula à metade. Logo, haverá log2n iterações de complexidade O(n).

Algoritmo 4: ATRIBUI POSIÇÕES AOS ELEMENTOS DA MATRIZ

```
1  Função atribuiPosicao(matriz[], Fp[],indice)
2      achou := false;
3      construtor := {⊨, ⊑, ⊓, ⊔, ¬, ∃, ∀};
4      para cada elemento i da matriz[] faça
5          se matriz[i] não é do tipo Elemento então
6              atribuiPosicao(matriz[i],Fp[],indice);
7          senão
               /* é um elemento                                    */
8              achou := false;
9              repita
10                 se Fp[indice].rotulo ∉ construtor então
11                     se Fp[indice].rotulo = matriz[i].literal então
12                         matriz[i].posicao := Fp[indice].posicao;
13                         achou := true;
14                         indice := indice + 1;
15                     senão
16                         indice := indice + 1;
17                 senão
18                     indice := indice + 1;
19             até (indice > tamanho de Fp[]) ou (achou = true);
20     retorna matriz;
```

Complexidade do algoritmo 4: m2 × n, logo O(m2 · n), dada as iterações da linha 4 e n iterações da linha 9.

Algoritmo 5: BUSCA CONEXÕES

```
1  Função buscaConexao(matriz[], C[], indice)
2      para cada elemento i da matriz[] faça
3          se matriz[i] não é do tipo Elemento então
4              buscaConexao(matriz[i], C[], indice);
5          senão
               /* se é diferente de null, então possui pelo menos uma conexão
                  */
6              se matriz[i].conexao[] ≠ null então
7                  para cada conexao j de matriz[i].conexao[] faça
8                      se constaConexao(C[],matriz[i].conexao[j].ordem) = false então
                           /* conexão ainda não consta no conjunto de conexões.
                              */
9                          se matriz[i].id = matriz[i].conexao[j].idelemento1 então
10                             C[indice].posicao1 := matriz[i].posicao;
11                             C[indice].posicao2 =
                                 buscaPosicao(matriz[i].conexao[j].idelemento2);
12                         senão
13                             C[indice].posicao2 := matriz[i].posicao;
14                             C[indice].posicao1 :=
                                 buscaPosicao(matriz[i].conexao[j].idelemento1);
15                         C[indice].ordem := matriz[i].conexao[j].ordem;
16                         indice := indice + 1;
       /* ordenar as conexões pelo campo ordem                                */
17     C[] := ordenaConexao(C[]);
18     retorna C;
```

Complexidade do algoritmo 5: O(m4), dada as m iterações da linha 2 e m iterações da linha 7, e as chamadas, dentro do escopo dessas iterações, das funções:
- buscaPosicao (ver algoritmo 13), linhas 11 e 14, com complexidade O(m2),
- constaConexao (ver algoritmo 12), linha 8, com complexidade O(c), temos a complexidade: m × m × (m2 + c) = m2 × (m2 + c) = m4 + m2 × c.
- ordenaConexao (ver algoritmo 14), na linha 17, tem complexidade: O(c2). Note que, a chamada para essa função está fora do escopo das iterações citadas acima. Assim: m4 + m2 × c + c2 = O(m4).

Algoritmo 6: GERA A ORDEM DE REDUÇÃO DAS POSIÇÕES

```
1  Função geraOrdemReducao(C, no)
2      para cada elemento i de C[] faça
           /* Busca o caminho para cada nó folha que forma a conexão      */
3          cn1[] = buscaCaminho(no, C[i].posicao1, ∅);
4          cn2[] = buscaCaminho(no, C[i].posicao2, ∅);
5          para cada caminho cnz faça
6              continua := true;
7              conectar := false;
8              j := 0;
9              repita
                   /* se o nó é do tipo folha                              */
10                 se cnz[j].tipo = null então
11                     empilhe cnz[j] no topo de P;
12                     continua := false;
13                     se z = 2 então
                           /* no 2º caminho e nó folha, então tentar conectar */
14                         conectar := true;
15                 senão
                       /* Se o nó cnz[j] não está no conj. R de nós que foram
                          reduzidos. Se ele está, não precisa mais inseri-lo. */
16                     se (cnz[j] ∉ R) então
17                         Executar Instruções do Algoritmo 7;
18                         j := j + 1;
19                     senão
20                         j := j + 1;
21             até continua = false;
22         se conectar = true então
23             Executar Instruções do Algoritmo 8;
24     retorna ◁;
```

Complexidade do algoritmo 6: O(n2), dado que c=2 × 2n + 2 × (n × n) + n2 = O(n2), onde, na sequência temos:

- c=2 é número de iterações da linha 2;
- 2n se refere as duas chamadas para buscaCaminho (ver algoritmo 15), nas linhas 3 e 4;
- 2 é o número máximo de iterações da linha 5;
- n é o número de iterações da linha 9;
- n, número de operações da linha 17, que está dentro do escopo da iteração da linha 9;
- n2, se refere as operações na linha 23, instruções do algoritmo 8.

Algoritmo 7: VERIFICA TIPO DA POSIÇÃO

```
1  se (cn_z[j].tipo = α ou β ou α') então
2      empilhe cn_z[j] em R;
3      empilhe cn_z[j] em em ◁;
4      se (cn_z[j].tipo = α') então
5          empilhe cn_z[j] em AlphaL;
6  senão se (cn_z[j].tipo = δ) então
7      empilhe cn_z[j] em Delta;
8      empilhe cn_z[j] em R;
9      empilhe cn_z[j] em ◁;
10     nosGamma[] := BUSCANOSTIPO(γ, P);
11     se nosGamma[] ≠ ∅ então
12         para cada nó gamma em nosGamma[] faça
13             σ_δ := SUBSTITUIPOSICAO(gamma, Delta, σ_Final);
14             empilhe nó gamma em R;
15             REMOVENO(gamma, P);
16             σ_Final := σ_δ;
17 senão se (cn_z[j].tipo = β') então
18     se AlphaL ≠ ∅ então
19         σ_β' := SUBSTITUIPOSICAO(cn_z[j], AlphaL, σ_Final);
20         empilhe cn_z[j] em R;
21         empilhe cn_z[j] em ◁;
22         σ_Final := σ_β';
23     senão
24         para cada nó no caminho cn_z a partir de j faça
25             se (CONSTANO(cn_z[j],P) = false) então
26                 empilhe cn_z[j] no topo de P;
27             j := j + 1;
28         continua := false;
29 senão se (cn_z[j].tipo = γ) então
30     se Delta ≠ ∅ então
31         σ_δ := SUBSTITUIPOSICAO(cn_z[j], Delta, σ_Final);
32         empilhe cn_z[j] em R;
33         σ_Final := σ_δ;
34     senão
35         para cada nó no caminho cn_z a partir de j faça
36             se (CONSTANO(cn_z[j],P) = false) então
37                 empilhe cn_z[j] no topo de P;
38             j := j + 1;
39         continua := false;
```

Complexidade do algoritmo 7: O(n), dado que todas as funções chamadas são de complexidade O(n), e estão dentro de estruturas de condição, sendo executadas apenas uma vez.

Algoritmo 8: CONECTAR

```
1  temp := SUBSTITUIPOSICAOFINAL(cn1[], cn2[], σFinal);
2  se temp ≠ null então
3      σFinal := temp;
4      resp := CHECAREFLEXIVIDADE(◁);
       /* Se resp = 0, então ◁ NÃO é reflexiva, caso contrário, é.      */
5      se (resp = 0) então
6          parLit[0]:= elemento do topo de cn1;
7          parLit[1]:= elemento do topo de cn2;
8          predicado := false;
           /* Relações são binárias, logo posSubst é igual a 2. Predicados,
              são unários, então posSubst = 1.                          */
9          se tamanho de parLit[0].posSubst = 1 então
10             predicado := true;
11         achou := false;
12         para (k := 0; k < 2, k++) faça
13             se (parLit[k] ∈ R) então
14                 achou := true;
15                 desempilhe elemento do topo de P;
16             senão
17                 empilhe o elemento do topo de P em R;
18                 desempilhe elemento do topo de P;
19         se (achou = false) e (predicado = true) então
               /* O par de literais que formam a conexão, parLit, são
                  armazenados em ◁. Isso fechará um ramo na estrutura da
                  prova parcial.                                        */
20             empilhe parLit em ◁;
21     senão
           /* ◁ é reflexiva, não é possível gerar prova em sequentes    */
22         retorna null;
23 senão
24     retorna null;
```

Complexidade do algoritmo 8: O(n2), dado que a única iteração é a da linha 12, que pode ser executada no máximo 2 vezes, e as chamadas para as funções abaixo, não estão dentro de estrutura de repetição:

- SubstituiPosicaoFinal (ver algoritmo 21): complexidade O(n2)
- checaReflexividade (ver algoritmo

Algoritmo 9: CONSTRÓI A ESTRUTURA DA PROVA (PARCIAL) EM SEQUENTES

```
1  Função constroiEstruturaSequentes(◁, idx)
2      i := idx;
3      se ◁[i] é um conjunto de nós então
4          retorna ◁[i];
5      senão
6          noEst := ◁[i];
7          noEst.direita := constroiEstruturaSequentes(◁, i + 1);
8          se noEst.tipo = β ou β' então
9              noEst.esquerda := constroiEstruturaSequentes(◁, i + 2);
10     retorna noEst;
```

Complexidade ao algoritmo 9: O(n2). A cada iteração, no pior caso, reduz-se a fórmula à metade. Portanto, haverá log2 n iterações de complexidade O(n).

Algoritmo 10: CONSTRÓI PROVA EM SEQUENTES

```
1  Função constroiSequentes(F, noArv, noEst, lado, regra, i, S)
2      se i>0 então
3          se regra = l⊓ ou r⊔ então
4              F := aplicaRegraL⊓R⊔(F, noEst);
5          senão se regra = r¬⊓ ou l¬⊔ então
6              F := aplicaRegraR¬⊓L¬⊔(F, noEst, noArv);
7          senão se regra = l¬¬ ou r¬¬ então
8              F := aplicaRegraL¬¬R¬¬(F, noEst, noArv);
9          senão se regra = l⊔ ou r⊓ ou l¬⊓ ou r¬⊔ então
10             F := aplicaRegraL⊔R⊓(F, noEst, lado);
11         senão se regra = r∀ ou l∃ ou l¬∀ ou r¬∃ então
12             F := aplicaRegraR∀L∃(F, noEst)
13         senão se regra = cut então
14             se noEst é um conjunto de nós então
15                 F := aplicaRegraCorte(F, noEst, lado, S);
16                 retorna F;
17             senão
18                 j := 0;
19                 enquanto S ≠ ∅ faça
20                     F[j]:= o elemento do topo de S;
21                     desempilhe o elemento do topo de S;
22                     j:= j +1;
23     se noEst é um conjunto de nós então
24         retorna null;
25     senão
26         regra := buscaRegraSequentes(noArv, noEst);
27         enquanto regra = 0 faça
28             regra := buscaRegraSequentes(noArv, noEst.direita);
29     se noEst.tipo = β' então
30         se noEst.direita é um conjunto de nós então
31             F.direita := constroiSequentes(F, noArv, noEst.direita, 'D', regra, 1, S);
32             F.esquerda := constroiSequentes(F,noArv,noEst.esquerda,'E', regra, 1, S);
33         senão
34             F.esquerda := constroiSequentes(F,noArv,noEst.esquerda,'E', regra, 1, S);
35             F.direita := constroiSequentes(F, noArv, noEst.direita, 'D', regra, 1, S);
36     senão
37         F.direita := constroiSequentes(F, noArv, noEst.direita,'D', regra, 1, S);
38     se noEst.tipo = β então
39         F.esquerda := constroiSequentes(F, noArv, noEst.esquerda, 'E', regra, 1, S);
40     retorna F;
```

Complexidade do algoritmo 10: O(n3), pois analisando a chamada de:

- aplicaRegraL⊓R⊔, na linha 4, tem O(n);
- aplicaRegraR¬ ⊓L¬ ⊔, na linha 6, O(n);

- aplicaRegraL¬ ¬R¬ ¬, na linha 8, tem O(n)
- aplicaRegraL⊔R⊓, na linha 10, tem O(n)
- aplicaRegraR∀L∃, na linha 12, tem O(n2), e
- aplicaRegraCorte, na linha 15, tem O(n).

Até aqui a complexidade é O(n2).

- buscaRegraSequentes, na linha 26, tem O(n)
- na linha 27, tem uma iteração para n=2 entradas, e
- buscaRegraSequentes, na linha 28, tem O(n), dentro do escopo da iteração da linha 27.

Logo, temos nesse outro ponto, O(n2). Como há uma recursividade, que reduz no pior caso a árvore pela metade, temos: log n recursões ×(O(n2) + O(n2)) = O(n3),

Algoritmo 11: Simula a exclusão dos parênteses da Fórmula Infixa e Atualiza Posições dos Elementos na variável global Pos

```
1  Função atualizaPosicao(F[])
2      int idx = 0;
3      int j;
4      para cada elemento m de F[] faça
5          se m ≠ '(' e m ≠ ')' então
6              idx:= idx + 1;
7              j = 0;
8              enquanto j < tamanho de pos[] faça
9                  j := j + 1;
10                 se pos[j] = posição atual de F[] então
11                     pos[j] := idx - 1;
12                     j := tamanho de pos[];
```

Complexidade do Algortimo 11: O(n), pois o algoritmo processa n entradas. As operações dentro do loop aumentam a complexidade em um fator constante, não alterando a complexidade assintótica.

Algoritmo 12: Checa se uma conexão consta no conjunto de conexões.

```
1  Função constaConexao(C[], ordem)
2      se C[] = ∅ então
3          retorna false;
4      senão
5          para cada elemento i em C[] faça
6              se C[i].ordem = ordem então
7                  retorna true;
8              senão
9                  retorna false;
```

Complexidade do Algortimo 12: O(c), onde c representa o tamanho de C[], ou seja, o número de conexões contidas no array. O algoritmo processa c entradas. As opera-

ções dentro do loop aumentam a complexidade em um fator constante, não alterando a complexidade assintótica.

Algoritmo 13: Busca a posição do elemento na matriz dado seu identificador

```
1  Função buscaPosicao(matriz[],idelemento)
2      achou := false;
3      int poselemento; i := 0;
4      enquanto (achou = false) ou (i <= tamanho de matriz[]) faça
5          se matriz[i] não é do tipo Elemento então
6              buscaPosicao(matriz[i], idelemento);
7          senão
8              se matriz[i].id = idelemento então
9                  poselemento := matriz[i].posicao;
10                 achou := true;
11             i := i + 1;
12     retorna poselemento;
```

Complexidade do Algortimo 13: O(m2). A cada iteração, no pior caso, reduz-se a fórmula à metade.

Algoritmo 14: Ordena conexões em ordem crescente pelo campo ordem

```
1  Função ordenaConexao(C[])
2      int n := tamanho de C[];
3      int min; ConexaoNo temp[];
4      para i = 0; i < n - 1; i++) faça
5          min := i;
6          para (j = i + 1; j < n; j++) faça
7              se C[j].ordem < C[min].ordem então
8                  min := j;
9          temp := C[i];
10         C[i] := C[min];
11         C[min] := temp;
12     retorna C;
```

Complexidade do Algortimo 14: O(c2), pois possui duas iterações aninhadas, com c entradas. c representa o tamanho de C[], ou seja, o número de conexões contidas no array.

Algoritmo 15: BUSCA O CAMINHO ENTRE O NÓ RAIZ E UM DADO NÓ

```
1  Função buscaCaminho(no, pos, caminho)
       /* Busca o caminho entre o nó raiz e um nó na árvore sintática     */
       /* O noNulo com posição igual a -1, indica a árvore vazia          */
2      No noNulo;
3      noNulo.pos = -1;
4      se no = null então
5          empilhe noNulo no topo do caminho;
6          retorna caminho;
7      senão
8          empilhe no.pos no caminho;
9          se no.pos = pos então
10             retorna caminho;
11         senão
12             caminho := buscaCaminho(no.direita, pos, caminho);
13             se elemento c do topo do caminho tem c.pos = −1 então
14                 desempilhe c do caminho;
15                 caminho := buscaCaminho(no.esquerda, pos, caminho);
16                 se elemento c do topo do caminho tem c.pos = −1 então
17                     desempilhe os dois primeiros elementos do topo do caminho;
18                     empilhe noNulo no topo do caminho;
19                     retorna caminho;
20                 senão
21                     retorna caminho;
22             senão
23                 retorna caminho;
```

Complexidade do algoritmo 15: O(n). Dado que, no pior caso, o algoritmo deve percorrer toda a árvore duas vezes, indo e voltando com o backtracking.

Algoritmo 16: SUBSTITUI POSIÇÕES σ_δ / $\sigma_{\beta'}$

```
1  Função substituiPosicao(no1, no2, σ)
2      par[0] := no1.posicao;
3      par[1] := no2.posicao;
4      se σ ≠ ∅ então
5          para cada elemento σ' em σ faça
6              se σ'[0] = no2.posicao então
7                  par[1] := σ'[1];
8      σ := σ ∪ par;
9      retorna σ;
```

Complexidade do algoritmo 16: O(n), pois a entrada para esse algoritmo é n=2 e só tem um loop de tamanho proporcional a entrada.

Algoritmo 17: CHECA SE UM NÓ CONSTA EM UMA LISTA DE NÓS.

```
1 Função constaNo(no, lista)
      /* Checa se um nó consta em uma lista de nós, como por exemplo na
         lista de nós que precisam ser reduzidos P                      */
2     para cada elemento i em lista faça
3         se lista[i] = no então
4             retorna true;
5         senão
6             retorna false;
```

Complexidade do algoritmo 17: O(n), pois tem no máximo tamanho proporcional a n iterações.

Algoritmo 18: BUSCA NÓS DE UM DADO TIPO EM UMA LISTA DE NÓS.

```
1 Função buscaNosTipo(tipo, lista)
      /* Procura nós em um array de acordo com um tipo, se encontrar
         informa quais são.                                             */
2     j:= 0;
3     v[];
4     para cada elemento i em lista faça
5         se lista[i].tipo = tipo então
6             v[j]:= lista[i];
7             j:= j + 1;
8     retorna v;
```

Complexidade ao algoritmo 18: O(n), pois tem no máximo tamanho proporcional a n iterações.

Algoritmo 19: REMOVE NÓ DE UMA LISTA.

```
1 Função removeNo(no, lista)
      /* Remove nó de um array.                                         */
2     j:= 0;
3     novaLista[];
4     para cada elemento i em lista faça
5         se lista[i] ≠ no então
6             novaLista[j]:= lista[i];
7             j:= j + 1;
8     retorna novaLista;
```

Complexidade ao algoritmo 19: O(n), pois tem no máximo tamanho proporcional a n iterações.

Algoritmo 20: CHECA SE ◁ É REFLEXIVA

```
1 Função checaReflexividade(◁)
2     resp := 0;
3     tamanho := tamanho de ◁;
4     para (i := 0; i < tamanho; i++) faça
5         para (j: = i + 1; j < tamanho; j++) faça
6             se (◁[j] = ◁[i]) então
7                 resp := 1; /* há um valor repetido, ◁ é reflexiva          */
8     retorna resp;
```

Complexidade do algoritmo 20: O(n2), pois tem duas iterações aninhadas com entrada de tamanho proporcional a n.

Algoritmo 21: SUBSTITUI POSIÇÕES σ_{Final}

```
1  Função substituiPosicaoFinal(cn1, cn2, σFinal)
2      no1:= elemento do topo de cn1;
3      no2:= elemento do topo de cn2;
4      desempilhe o elemento do topo de cn1;
5      desempilhe o elemento do topo de cn2;
6      paiNo1:= elemento do topo de cn1;
7      paiNo2:= elemento do topo de cn2;
       /* tamanho de no1.posSubst[] é no máximo 2, para nós que são relações
          */
8      tam := tamanho de no1.posSubst[];
9      para (i = 0, i < tam, i++) faça
10         par[0] := no1.posSubst[i];
11         par[1] := no2.posSubst[i];
           /* se posições ou instâncias associadas ao rótulo Não são
              idênticas, busca se essas posições já foram substituídas; se
              são idênticas, não faz nada, nem mesmo insere em σFinal      */
12         se (par[0] ≠ par[1]) então
13             se σFinal ≠ ∅ então
14                 para cada elemento σ'Final em σFinal faça
15                     se σ'Final[0] = no1.posSubst[i] então
16                         par[0] := σ'Final[1];
17                     senão se σ'Final[0] = no2.posSubst[i] então
18                         par[1] := σ'Final[1];
19         se (par[0] ≠ par[1]) então
               /* faz a substituição da posição por uma instância se um dos
                  nós é filho de nó do tipo α ou α', caso contrário retorna
                  null, indicando conexão não complementar                  */
20             se paiNo1.tipo = α ou α' então
21                 par[0] := par[1];
22                 par[1] := no1.posSubst[i];
23                 σFinal := σFinal ∪ par;
24             senão se paiNo2.tipo = α ou α' então
25                 σFinal := σFinal ∪ par;
26             senão
27                 retorna null;
28     retorna σFinal;
```

Complexidade ao algoritmo 21: O(n2).

Algoritmo 22: APLICA REGRA L⊓ OU R⊔ SOBRE F.

```
1  Função aplicaRegraL⊓R⊔(noArv, noEst)
2      achou := false;
3      i := 0;
4      j := 0
5      idx := buscaIndice(F,noEst.posicao);
6      par := posicaoParenteses(F,idx);
7      enquanto i < tam faça
8          se (i = par[0]) ou (i = par[1]) então
9              i := i + 1;
10         senão se F.No[i].posicao = noEst.posicao então
11             auxF[j].rotulo := ',';
12             auxF[j].posicao := noEst.posicao;
13             j := j + 1;
14             i := i + 1;
15         senão
16             auxF[j] := F.No[i];
17             j := j + 1;
18             i := i + 1;
19     retorna F;
```

Complexidade do algoritmo 22: O(n), pois tem no máximo n iterações na linha 7. As funções chamadas nas linhas 5 e 6, estão fora do escopo da iteração, e ambas possuem complexidade O(n).
Assim: O(n) + O(n) + O(n) = O(n).

Algoritmo 23: Aplica regra R¬⊓ ou L¬⊔ sobre F.

```
1  Função aplicaRegraR¬⊓L¬⊔(F, noEst, noArv)
2      achou := false;
3      i := 0; j := 0;
4      idx := buscaIndice(F,noEst.posicao);
5      cn1[] = buscaCaminho(noArv, noEst.posicao, ∅);
6      noNeg := buscaNoRotulo(¬, cn1[]);
7      par := posicaoParenteses(F,idx);
8      enquanto i < tam faça
9          se (i = par[0]) ou (i = par[1]) então
10             i := i + 1;
11         senão se F.No[i].posicao = noEst.posicao então
12             auxF[j].rotulo := ',';
13             auxF[j].posicao := noEst.posicao;
14             j := j + 1;
15             auxF[j].rotulo := noNeg.rotulo;
16             auxF[j].posicao := noNeg.posicao;
17             j := j + 1;
18             i := i + 1;
19         senão
20             auxF[j] := F.No[i];
21             j := j + 1;
22             i := i + 1;
23     F.No[]:= auxF;
24     retorna F;
```

Complexidade do algoritmo 23: O(n), pois tem no máximo n iterações. As funções chamadas nas linhas 5, 6, 7 e 8, estão fora do escopo da iteração, e todas possuem complexidade O(n). Assim: O(n) + O(n) + O(n) + O(n) + O(n) = O(n).

Algoritmo 24: Aplica regra L¬¬ ou R¬¬ sobre F.

```
1  Função aplicaRegraL¬¬R¬¬(F, noEst, noArv)
2      achou := false;
3      j := 0;
4      cn1[] = buscaCaminho(noArv, noEst.posicao, ∅);
5      noNeg := buscaNoRotulo(¬, cn1[]);
6      para (i:=0; i<tam; i++) faça
7          se (F.No[i].posicao = noNeg.posicao) e (F.No[i].posicao = noEst.posicao)
           então
8              auxF[j] := F.No[i];
9              j := j + 1;
10     F.No[]:= auxF;
11     retorna F;
```

Complexidade do algoritmo 24: O(n), pois tem no máximo n iterações. As funções chamadas nas linhas 4, e 5, estão fora do escopo da iteração, e todas possuem complexidade

O(n).
Assim: O(n) + O(n) + O(n) = O(n).

Algoritmo 25: Aplica regra (L⊔ ou R⊓) ou (L¬⊓ ou R¬⊔) sobre F.

```
1  Função aplicaRegraL⊔R⊓(F, noEst, lado)
2      par[];
3      achou := false;
4      i := 0;
5      j := 0;
6      idx := buscaIndice(F,noEst.posicao);
7      par := posicaoParenteses(F,idx);
8      se lado = 'd' então
9          enquanto i < tam faça
10             se (i >= par[0] e i <= idx) ou (i = par[1] ) então
11                 i := i + 1;
12             senão
13                 auxF[j] := F.No[i];
14                 j := j + 1;
15                 i := i + 1;
16     se lado = 'e' então
17         enquanto i < tam faça
18             se (i >= idx e i <= par[1]) ou (i = par[0] ) então
19                 i := i + 1;
20             senão
21                 auxF[j] := F.No[i];
22                 j := j + 1;
23                 i := i + 1;
24     F.No[]:= auxF;
25     retorna F;
```

Complexidade do algoritmo 25: O(n), pois as iterações das linhas 9 e 17 são exclusivas, e tem no máximo n iterações. As funções chamadas nas linhas 6, e 7, estão fora do escopo das iterações, e todas possuem complexidade O(n).
Assim: O(n) + O(n) + O(n) = O(n).

Algoritmo 26: APLICA REGRA (R∀ OU L∃) OU (L¬∀ OU R¬∃) SOBRE F.

```
1  Função aplicaRegraR∀L∃(F, noEst)
       /* armazena as posições de todos os quantificadores que devem ser
          reduzidos juntos.                                              */
2      posQuants[] := noEst.posNRJ[];
3      tam := tamanho de posQuants[];
4      posQuants[tam] := noEst.posicao;
5      tam := tamanho de posQuants[];
6      j :=0;
7      tam := tamanho de F.No[];
8      t := tamanho de idx[];
9      i := 0;
10     enquanto i<tam faça
11         para k := 0; k<t; k++ faça
12         se F.No[i].posicao = posQuants[k] então
13             i := i + 2;
               /* para 'eliminar' a relação associada ao quantificador.  */
14         senão
15             auxF[j] := F.No[i];
16             j := j + 1;
17     F.No[]:= auxF;
18     retorna F;
```

Complexidade do algoritmo 26: O(n2), pois existem duas iterações aninhadas e executadas em um número proporcional a n. A primeria, na linha 10 tem entrada n, a segunda, na linhas 11, entrada n=2.
Assim, a complexidade é n × n=2 = O(n2).

Algoritmo 27: APLICA REGRA DO CORTE SOBRE F.

```
1  Função aplicaRegraCorte(F, noEst, lado, S)
2      continua := true;
3      cont := 0;
       /* O nó noEst é um array com os 2 nós que formaram a conexão. O nó
          com menor posição, faz parte do axioma inicial.                  */
4      se noEst[0].posicao < noEst[1].posicao então
5          idNoAxioma := noEst[0].posicao;
6          idNo := noEst[1].posicao;
7      senão
8          idNoAxioma := noEst[1].posicao;
9          idNo := noEst[0].posicao;
10     se lado = 'D' então
           /* temos o axioma inicial para o ramo direito do sequentes      */
11         axioma := buscaSequenteAxiomaD(F, idNoAxioma);
           /* temos o consequente do sequente a ser provado                */
12         sucedente := buscaAntecedenteSequente(axioma, idNoAxioma)
           /* temos um sequente do consequente da fórmula. Falta extrair seu
              antecedente.                                                 */
13         seqCons := buscaSequenteAxiomaD(F, idNo);
           /* temos o antecedente do sequente a ser provado                */
14         antecedente := buscaAntecedenteSequente(seqCons, idNoAxioma)
15     senão
           /* temos o axioma inicial para o ramo esquerdo do sequentes     */
16         axioma := buscaSequenteAxiomaE(F, idNoAxioma);
           /* temos o antecedente do sequente a ser provado                */
17         antecedente := buscaConsequenteSequente(sequente)
           /* temos um sequente do consequente da fórmula. Falta extrair seu
              sucedente.                                                   */
18         seqCons := buscaSequenteAxiomaE(F, idNo);
           /* temos o sucedente do sequente a ser provado                  */
19         sucedente := buscaConsequenteSequente(sequente)
       /* O sequente a ser provado é a junção: antecedente + → + sucedente */
       /* S recebe seus elementos na ordem da direita para esquerda.       */
20     enquanto sucedente ≠ ∅ faça
21         empilhe o elemento do topo de sucedente em S;
22         desempilhe o elemento do topo de sucedente;
23     empilhe '→' em S;
24     enquanto antecedente ≠ ∅ faça
25         empilhe o elemento do topo de antecedente em S;
26         desempilhe o elemento do topo de antecedente;
27     F.No[]:= axioma;
28     retorna F;
```

Complexidade do algoritmo 27: O(n), pois todas as chamadas das funções listadas abaixo tem complexidade O(n) :

- buscaSequenteAxiomaD, na linha 10,
- buscaAntecedenteSequente, na linha 11,
- buscaSequenteAxiomaD, na linha 12,

- buscaAntecedenteSequente, na linha 13,
- buscaSequenteAxiomaE, na linha 15,
- buscaConsequenteSequente, linha 16,
- buscaSequenteAxiomaE, linha 17,
- buscaConsequenteSequente, linha 18,
- e as iterações nas linhas 19 e 23 são independentes, e cada uma tem complexidade O(n).

Assim: O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) + O(n) = O(n).

Algoritmo 28: BUSCA UM SEQUENTE OU AXIOMA INICIAL, DA DIREITA PARA ESQUERDA DA DADA FÓRMULA

```
1  Função buscaSequenteAxiomaD(F, idNo)
       /* pega o índice de ')' na fórmula que delimita o axioma inicial ou
          sequente que estamos procurando. Essa busca é feita da direita
          para a esquerda.                                              */
2      idx := buscaIndice(F,idNo);
3      enquanto continua = true faça
4          idx := idx + 1 ;
5          se F.No[idx].rotulo = ')' então
6              cont := cont + 1;
7          senão
8              continua := false;
9      idx := idx - 1;
10     continua := true; cont := 0;
       /* empilha em auxF o axioma inicial                              */
11     enquanto continua = true faça
12         se F.No[idx].rotulo = ')' então
13             cont := cont + 1;
14         senão se F.No[idx].rotulo = '(' então
15             cont := cont - 1;
16             se cont = 0 então
17                 continua = false;
18         empilhe em F.No[idx] em auxF;
19         idx := idx - 1 ;
       /* o axioma agora é emplilhado em axioma, sentido esquerda - direita
          */
20     enquanto auxF ≠ ∅ faça
21         empilhe o elemento do topo de auxF em axioma;
22         desempilhe o elemento do topo de auxF;
23 retorna axioma
```

Complexidade do algoritmo 28: O(n), pois:

- buscaIndice, na linha 2, tem complexidade O(n),
- e as iterações nas linhas 3, 11 e 20 não são aninhadas, e cada uma tem complexidade O(n).

Assim: O(n) + O(n) + O(n) = O(n).

Algoritmo 29: Busca um Sequente ou Axioma Inicial, da esquerda para direita da dada fórmula

```
1  Função buscaSequenteAxiomaE(F, idNo)
       /* pega o índice de '(' na fórmula que delimita o axioma inicial ou
          sequente que estamos procurando. Essa busca é feita da esquerda
          para direita.                                                  */
2      idx := buscaIndice(F,idNo);
3      enquanto continua = true faça
4          idx := idx - 1 ;
5          se F.No[idx].rotulo = '(' então
6              cont := cont + 1;
7          senão
8              continua := false;
9      idx := idx + 1;
10     continua := true; cont := 0;
       /* empilha em auxF o axioma inicial                               */
11     enquanto continua = true faça
12         se F.No[idx].rotulo = '(' então
13             cont := cont + 1;
14         senão se F.No[idx].rotulo = ')' então
15             cont := cont - 1;
16             se cont = 0 então
17                 continua = false;
18         empilhe em F.No[idx] em auxF;
19         idx := idx + 1 ;
       /* o axioma recebe auxF, que já está no sentido esquerda - direita
          (último elemento da direita = elemento do topo da pilha)       */
20     axioma := auxF;
21 retorna axioma
```

Complexidade do algoritmo 29: O(n), pois as iterações nas linhas 3 e 11, não são aninhadas e cada uma pode realizar n iterações.
Assim: O(n) + O(n) = O(n).

Algoritmo 30: BUSCA O ANTECEDENTE E UM SEQUENTE/AXIOMA INICIAL

```
1  Função buscaAntecedenteSequente(sequente, idNo)
       /* Armazena o antecedente de um Sequente/axioma. Para isso, é
          preciso desempilhar os elementos até o → e contar os ')'      */
2      continua := true;
3      cont := 0;
4      enquanto continua = true faça
5          se o atributo posicao do elemento do topo de sequente = idNo então
6              empilhe o elemento do topo de sequente em temp;
7              desempilhe o elemento do topo de sequente;
8              continua := false;
9          empilhe o elemento do topo de sequente em temp;
10         desempilhe o elemento do topo de sequente;
       /* Retira os parênteses excedentes                               */
11     cont := 0;
12     enquanto continua = true faça
13         se o atributo rotulo do elemento do topo de sequente = ')' então
14             cont := cont + 1;
15             empilhe o elemento do topo de sequente em auxSeq;
16             desempilhe o elemento do topo de sequente;
17         senão se o atributo rotulo do elemento do topo de sequente = '(' então
18             se cont > 1 então
19                 cont := cont - 1;
20                 empilhe o elemento do topo de sequente em auxSeq;
21                 desempilhe o elemento do topo de sequente;
22             senão se cont = 0 então
23                 continua = false;
24     enquanto auxSeq ≠ ∅ faça
25         empilhe o elemento do topo de auxSeq em antecedente;
26         desempilhe o elemento do topo de auxSeq;
27 retorna antecedente;
```

Complexidade do algoritmo 30: O(n), pois as iterações nas linhas 4, 12 e 24, não são aninhadas e cada uma pode realizar n iterações.
Assim: O(n) + O(n) + O(n) = O(n).

Algoritmo 31: BUSCA O CONSEQUENTE DE UM SEQUENTE/AXIOMA INICIAL

```
1  Função buscaConsequenteSequente(sequente)
       /* Armazena o consequente de um Sequente/axioma. Para isso, é
          preciso desempilhar os elementos após o → e contar os '('      */
2      continua := true;
3      cont := 0;
4      consequent;
       /* desempilha todos os elementos que antecedem →                  */
5      enquanto o rotulo do elemento do topo de sequente ≠ '→' faça
6          desempilhe o elemento do topo de sequente;
       /* desempilha o →                                                 */
7      desempilhe o elemento do topo de sequente;
       /* Descobre o consequente, 'eliminando' os parênteses excedentes  */
8      cont := 0;
9      enquanto continua = true faça
10         se o atributo rotulo do elemento do topo de sequente = '(' então
11             cont := cont + 1;
12             empilhe o elemento do topo de sequente em consequente ;
13             desempilhe o elemento do topo de sequente;
14         senão se o atributo rotulo do elemento do topo de sequente = ')' então
15             se cont > 1 então
16                 cont := cont - 1;
17                 empilhe o elemento do topo de sequente em consequente ;
18                 desempilhe o elemento do topo de sequente;
19             senão se cont = 0 então
20                 continua = false;
21         senão
               /* o atributo rotulo do elemento do topo de sequente é um
                  literal                                                */
22             empilhe o elemento do topo de sequente em consequente ;
23             desempilhe o elemento do topo de sequente;
24 retorna consequente
```

Complexidade do algoritmo 31: O(n), pois as iterações nas linhas 5 e 9, não são aninhadas e cada uma pode realizar n iterações.
Assim: O(n) + O(n) = O(n).

Algoritmo 32: BUSCA O ÍNDICE DE UM NÓ EM UM ARRAY.

```
1 Função buscaÍndice(F,posicao)
    /* Busca índice de nó em um array                    */
2   tam := tamanho de F;
3   i := 0;
4   enquanto (i < tam) e (achou = false) faça
5     se F.No[i].posicao = posicao então
6       achou := true;
7       idx := i;
8     i := i + 1;
9   retorna idx
```

Complexidade do algoritmo 32: O(n), pois realiza um tamanho proporcional a n iterações.

Algoritmo 33: ENCONTRA POSIÇÕES DOS PARÊNTESES QUE DELIMITAM A ABRANGÊNCIA DE UM CONECTIVO.

```
1  Função posicaoParenteses(F,idx)
2    achou := false;
3    tam := tamanho de F.No[];
4    j := 0;
5    i := idx - 1;
     /* Procura o '(' mais próximo do nó de índice idx.  */
6    enquanto (i >= 0) e (achou = false) faça
7      se F.No[i].rotulo = '(' então
8        achou := true;
9        par[j] := i;
10       j:= j + 1;
11     i := i + 1;
12   achou := false;
13   i := idx + 1;
14   enquanto (i < tam) e (achou = false) faça
15     se F.No[i].rotulo = ')' então
16       achou := true;
17       par[j] := i;
18       j:= j + 1;
19     i := i + 1;
20   retorna par;
```

Complexidade do algoritmo 33: O(n), pois as iterações nas linhas 6 e 14, não são aninhadas e cada uma realiza um tamanho proporcional a n iterações.
Assim: O(n) + O(n) = O(n).

Algoritmo 34: Busca a regra no cálculo de sequentes correspondente ao nó.

```
1 Função buscaRegraSequentes(noArv, noEst)
2     cn1[] = buscaCaminho(noArv, noEst.posicao, ∅);
3     resp := constaNoRotulo(¬, cn1[]);
      /* Se a negação precede o nó, busca a regra em uma tabela X, que deve
         representar a tabela 8, senão, na tabela Y, representando a
         tabela 7.                                                        */
4     se resp = true então
5         regra := buscaRegra(noEst, tabela X);
6     senão
7         regra := buscaRegra(noEst, tabela Y);
8     retorna regra;
```

Complexidade do algoritmo 34: O(n), pois:
- buscaCaminho, na linha 2, tem complexidade O(n);
- constaNoRotulo, na linha 3, tem complexidade O(n);
- as linhas 5 e 7, são buscas em tabelas de tamanho fixo.

Assim: O(n) + O(n) + O(1) + O(1) = O(n).

Algoritmo 35: Checa um nó com um dado rótulo está no caminho entre o nó raiz e um dado nó.

```
1 Função constaNoRotulo(rotulo, caminho)
      /* Checa um nó com um dado rótulo está no caminho entre o nó raiz e
         um dado nó. Por exemplo, um nó com rótulo igual a ¬ precede um
         certo nó.                                                        */
2     achou := false;
3     i := 0;
4     tam := tamanho do caminho;
5     enquanto (i <= tam - 2) e (achou = false) faça
6         se caminho[i].rotulo = rotulo então
7             achou := true;
8     retorna achou;
```

Complexidade do algoritmo 35: O(n), pois ele realiza um tamanho proporcional a n iterações.

Algoritmo 36: BUSCA UM NÓ COM UM DADO RÓTULO NO CAMINHO ENTRE O NÓ RAIZ E UM DADO NÓ.

```
1 Função buscaNoRotulo(rotulo, caminho)
      /* Busca um nó com um dado rótulo no caminho entre o nó raiz e um
         dado nó. Por exemplo, um nó com rótulo igual a ¬ precede um certo
         nó.                                                              */
2     achou := false;
3     i := 0;
4     tam := tamanho do caminho;
5     enquanto (i <= tam - 2) e (achou = false) faça
6         se caminho[i].rotulo = rotulo então
7             achou := true;
8     retorna caminho[i];
```

Complexidade do algoritmo 36: O(n), pois ele realiza um tamanho proporcional a n iterações.