UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

INTERPRETADOR DE FÓRMULAS DO CÁLCULO PROPOSICIONAL

MICHELE MILANE TAMBOSI

BLUMENAU
2003

2003/2-30

MICHELE MILANE TAMBOSI

INTERPRETADOR DE FÓRMULAS DO CÁLCULO PROPOSICIONAL

Trabalho de Conclusão de Curso submetido à Universidade Regional de Blumenau para a obtenção dos créditos na disciplina Trabalho de Conclusão de Curso II do curso de Ciência da Computação — Bacharelado.

Prof. Joyce Martins – Orientadora

BLUMENAU
2003

2003/2-30

INTERPRETADOR DE FÓRMULAS DO CÁLCULO PROPOSICIONAL

Por

MICHELE MILANE TAMBOSI

Trabalho aprovado para obtenção dos créditos na disciplina de Trabalho de Conclusão de Curso II, pela banca examinadora formada por:

Presidente:	Prof. Joyce Martins – Orientadora, FURB
Membro:	Prof. Paulo de Tarso Mendes Luna, FURB
Membro:	Prof. Jomi Fred Hübner, FURB

Blumenau, 10 de dezembro de 2003

Dedico este trabalho aos meus pais, pelo amor, dedicação e apoio que me deram para que eu chegasse até aqui.

AGRADECIMENTOS

Agradeço primeiramente à minha orientadora Joyce Martins, pela orientação e dedicação na condução deste trabalho.

Ao meu namorado Cleyton, pelo carinho e compreensão dedicados nesta etapa da minha vida.

Aos meus irmãos Mary e Jackson, pelo incentivo e força nas horas de desânimo.

E a todos que participaram da realização deste trabalho.

RESUMO

Este trabalho apresenta o desenvolvimento de um interpretador de fórmulas do cálculo proposicional que pode ser utilizado pelos acadêmicos da disciplina de Lógica para Computação do Curso de Ciências da Computação da Universidade Regional de Blumenau, com o objetivo de facilitar a compreensão da teoria estudada. O acadêmico poderá utilizar a ferramenta para escrever fórmulas ou argumentos do cálculo proposicional, verificando a sintaxe e validade dos mesmos. Para o desenvolvimento desta ferramenta foram empregados princípios e técnicas de construção de compiladores.

Palavras chaves: Lógica; Cálculo Proposicional; Interpretadores.

ABSTRACT

This work presents the development of a formula interpreter of the propositional calculation that can be used by the academics of disciplines of Logic for Computation of the Computer Science Course of the Universidade Regional de Blumenau, with the objective to facilitate the understanding of the studied theory. The academic will be able to use the tool to write formulas or arguments of the propositional calculation, verifying the syntax and validity of the same ones. For the development of this tool, principles and techniques of compilers construction had been used.

Keys Words: Logic; Propositional calculation; Interpreters.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE SIGLAS

fbf - fórmula bem-formada

BNF - *Backus Naur Form*

UML - *Unified Modeling Language*

RAD - Desenvolvimento Rápido de Aplicações

GALS – Gerador de Analisadores Léxicos e Sintáticos

SUMÁRIO

1 INTRODUÇÃO

Dado o seguinte problema, que pode ser considerado como uma atividade de raciocínio do cotidiano:

> Você foi convocado a participar do júri em um processo criminal. O advogado de defesa argumenta o seguinte: Se meu cliente fosse culpado, a faca estaria na gaveta. Ou a faca não estava na gaveta ou Jacson Pritchard viu a faca. Se a faca não estava lá no dia 10 de outubro, então Jacson Pritchard não viu a faca. Além disso, se a faca estava lá no dia 10 de outubro, então a faca estava na gaveta e o martelo estava no celeiro. Mas todos sabemos que o martelo não estava no celeiro. Portanto, senhoras e senhores, meu cliente é inocente. (GERSTING, 2001, p. 1)

Pergunta-se: o cliente é inocente?

Gersting (2001) afirma que se este argumento estivesse escrito com a notação da lógica formal, seria mais fácil responder a esta pergunta. Na verdade, o uso da lógica na representação dos processos de raciocínio pode ser aplicado em diversas áreas de conhecimento como psicologia, direito, filosofia, computação, etc.

A lógica para computação fundamenta-se no conceito de lógica formal. Várias são as definições encontradas. Ferreira (1975, p.1044) define lógica como “coerência de raciocínio, de idéia; seqüência coerente, regular e necessária de acontecimentos, de coisas”. Para Mortari (2001, p. 2), lógica “é a ciência que estuda princípios e métodos de inferência, tendo o objetivo principal de determinar em que condições certas coisas se seguem (são conseqüência), ou não de outras”.

A lógica é uma área que tem sido muito discutida e disseminada nos dias atuais, porém não é nova. O estudo das condições em que se pode afirmar que um dado raciocínio é correto, foi desenvolvido por filósofos como Parmênides, Sócrates e Platão. No entanto, foi Aristóteles quem sistematizou e definiu a lógica como atualmente é conhecida, constituindo-a como uma ciência autônoma. Com Aristóteles (384 a.C. - 322 a.C.) é que se dá o verdadeiro nascimento da lógica, ciência das idéias e dos processos da mente (FONSECA, 1998).

Conforme Fonseca (1998), em meados do século XIX houve uma verdadeira revolução no estudo da lógica. Ela passou a ser vista como um cálculo, tal como a álgebra, já que ambas são leis do pensamento humano. A George Boole (1815-1864) é atribuída a criação da lógica matemática que, pela primeira vez, de uma forma consistente, tratou a lógica como um cálculo.

No final do século XIX, os estudos da lógica matemática deram passos gigantescos, no sentido da formalização dos conceitos e processos demonstrativos. Entre os matemáticos e

filósofos que mais contribuíram para os avanços, destacam-se Gottlob Frege, Peano, Bertrand Russell, Alfred N. Witehead e David Hilbert. É nesta fase que são criados os sistemas lógicos: o cálculo proposicional e o cálculo de predicados (FONSECA, 1998).

O cálculo proposicional e o cálculo dos predicados são subconjuntos da lógica matemática. O primeiro é uma lógica mais simples e consiste na formalização e estudo de conectivos, enquanto o segundo é uma extensão da lógica das proposições em que se consideram variáveis e quantificadores sobre as variáveis.

Fonseca (1998) afirma que ao longo do século XX assistiu-se a generalização e diversificação dos estudos da lógica matemática, atingindo um elevado grau de formalização. A lógica possui atualmente um sistema completo de símbolos e regras de combinação de símbolos para obter conclusões válidas. Este fato tornou-a adapta a ter aplicações diretas em ciência da computação, tais como:

a) elaboração de programas: aplicada na construção de software por programadores;
b) representação do conhecimento: aplicada na inteligência artificial em sistemas especialistas;
c) manipulação de expressões lógicas: aplicada na simplificação de expressões em comandos de seleção e repetição;
d) teoria da recursão: trata do que pode e do que não pode efetivamente ser computável (máquina de Turing);
e) formalização de linguagens: aplicada no estudo da sintaxe e a semântica de construções lingüísticas;
f) projeto e análise de circuitos digitais: aplicada na eletrônica digital onde todas as tensões de entrada e de saída serão baixas (F=0) ou altas (T=1).

Embora o cálculo proposicional seja a lógica mais simples, alguns acadêmicos apresentam dificuldades no seu aprendizado. Existem várias ferramentas que utilizam o cálculo proposicional, algumas destas podem ser vistas em Mendes (2003). No entanto estas ferramentas não são adequadas para serem usadas no Curso de Ciências da Computação da Universidade Regional de Blumenau (FURB) em função da abordagem dada no ensino do cálculo proposicional na disciplina de Lógica para Computação. Portanto, este trabalho tem como objetivo construir um interpretador de fórmulas do cálculo proposicional, que permita verificar se uma fórmula está correta, motivando a compreensão dos conceitos envolvidos e

permitindo que os acadêmicos possam relacionar teoria e prática, na disciplina de Lógica para Computação do Curso de Ciências da Computação da Universidade Regional de Blumenau.

1.1 OBJETIVOS DO TRABALHO

O objetivo do presente trabalho é o desenvolvimento de um interpretador de fórmulas do cálculo proposicional para o uso como ferramenta de apoio na disciplina de Lógica para Computação do Curso de Ciências da Computação da FURB.

Os objetivos específicos do trabalho são:

a) verificar a sintaxe das fórmulas;
b) verificar as propriedades semânticas das fórmulas, visto que uma fórmula pode ser tautologia, contraditória ou satisfatível;
c) verificar a sintaxe dos argumentos;
d) verificar a validade dos argumentos.

1.2 ESTRUTURA DO TRABALHO

O trabalho está organizado em cinco capítulos. O capítulo seguinte apresenta a evolução dos conceitos de lógica, mais especificamente o cálculo proposicional, incluindo a descrição da sintaxe e da semântica de fórmulas e argumentos.

O terceiro capítulo trata de processadores de linguagens, apresentando tipos de tradutores de linguagens de programação e a estrutura modular de um tradutor. É dada ênfase aos módulos que foram implementados para a construção da ferramenta proposta.

O desenvolvimento do interpretador de fórmulas do cálculo proposicional é apresentado no capítulo quatro, sendo que a especificação, a implementação e o funcionamento da ferramenta desenvolvida são relatados.

Por fim, no capítulo cinco encontram-se as conclusões provenientes da execução desse trabalho e as possíveis extensões que podem ser desenvolvidas.

2 LÓGICA

O que é lógica? O que significa estudar lógica? Qual a sua definição? Para melhor entender a lógica, neste capítulo será inicialmente apresentada a evolução dos seus conceitos, que está dividida em três períodos: clássico (século IV a.C ao século XIX), moderno (século XIX a princípios do século XX) e contemporâneo (início do século XX até os dias atuais).

2.1 EVOLUÇÃO DOS CONCEITOS

Segundo Fonseca (1998), a lógica foi considerada na tradição clássica e medieval como instrumento indispensável ao pensamento científico. Atualmente é parte importante na metodologia dedutiva das ciências, além de constituir-se como um saber próprio, com abertura a relevantes problemas teóricos. Da ciência lógica nasceu a lógica matemática e, dentro desta, várias filosofias da lógica que interpretam os cálculos simbólicos e sua sistematização axiomática.

Para melhor compreender a lógica, a evolução dos seus conceitos será estudada, sendo dividida em três períodos: clássico, moderno e contemporâneo.

2.1.1 Período Clássico

> O início da ciência da Lógica encontra-se na antiga Grécia. As polêmicas geradas pela teoria de Parmênides e os famosos argumentos de Zenão de Eleia, que negavam a realidade do movimento fazendo um uso indevido do princípio da não-contradição, contribuíram para a distinção dos conceitos. Mais tarde, Sócrates defendeu o valor dos conceitos e tentou defini-los com precisão. Assim a Lógica como ciência vai se formando pouco a pouco, principalmente com Sócrates e Platão (FONSECA, 1998).

Mas, segundo Mortari (2001), é com Aristóteles (384 a.C. - 322 a.C.) que se dá efetivamente o surgimento do estudo da lógica. Foram múltiplas as contribuições de Aristóteles para a criação e desenvolvimento da lógica como é conhecida hoje. Entre outras, devem-se as seguintes contribuições, conforme Fontes (2003):

a) a separação da validade formal do pensamento e do discurso da sua verdade material;
b) a identificação dos conceitos básicos da lógica;
c) a introdução de letras mudas para denotar os termos;
d) a criação de termos fundamentais para analisar a lógica do discurso: Válido, Não Válido, Contraditório, Universal, Particular.

A lógica de Aristóteles tinha um objetivo eminentemente metodológico. Tratava-se de mostrar o caminho correto para a investigação, o conhecimento e a demonstração científicas.

Apesar dos enormes avanços que produziu, a lógica aristotélica tinha muitas limitações que se revelaram mais tarde, verdadeiros obstáculos para o avanço da ciência.

Embora Aristóteles seja o mais brilhante e influente filósofo grego, outra importante tradição argumentativa formou-se na antiga Grécia, com os megáricos e estóicos. Eles desenvolveram uma teoria lógica diferente da de Aristóteles, tão importante quanto a dele. Hoje essa teoria forma a base da lógica proposicional.

Segundo Fonseca (1998), a lógica durante a Idade Média era entendida como a "ciência de todas as ciências". Ainda nesta época deu-se uma enorme importância à dedução, desvalorizando-se por completo a indução na descoberta científica.

2.1.2 Período Moderno

A lógica moderna começou no século XVII com o filósofo e matemático alemão Gottfried Wilhelm Leibniz. Seus estudos influenciaram, 200 anos mais tarde, vários ramos da lógica matemática moderna e outras áreas relacionadas. Este filósofo procurou aplicar à lógica o modelo de cálculo algébrico da sua época. Esta é concebida como um conjunto de operações dedutivas de natureza mecânica onde são utilizados símbolos técnicos. Era sua intenção submeter a estes cálculos algébricos a totalidade do conhecimento científico.

Em meados do século XIX, diversos investigadores de formação matemática conceberam, não apenas uma nova linguagem simbólica, mas também uma forma de transformar a lógica numa álgebra. O inglês George Boole (1815-1864) é considerado o fundador da lógica matemática. Ele desenvolveu com sucesso o primeiro sistema formal para raciocínio lógico. Mais ainda, Boole foi o primeiro a enfatizar a possibilidade de se aplicar o cálculo formal a diferentes situações, e fazer operações com regras formais, desconsiderando noções primitivas.

No final do século XIX o estudo da lógica teve grande avanço com os filósofos e matemáticos como Frege, Peano, Russell, Witehead e Hilbert. Foi nesta fase que foram criados os sistemas lógicos: o cálculo proposicional e o cálculo de predicados, formulados inicialmente por Frege (1848-1925). Ele procurou em síntese criar todo um sistema capaz de transformar em raciocínios dedutivos todas as demonstrações matemáticas (MORTARI, 2001).

Giuseppe Peano (1858-1932), com base no trabalho de Frege, desenvolveu o sistema de notação usado pelos lógicos e matemáticos. Peano demonstrou igualmente que os enunciados matemáticos não são obtidos por intuição, mas sim deduzidos a partir de premissas.

A partir de contatos com a obra de Frege e adaptações da obra de Peano, o filósofo Bertrand Russel (1872-1970) procurou desenvolver o projeto do logicismo, isto é, a redução das matemáticas à lógica. A sua volumosa obra "Principia Mathematica" (1910-1913), escrita em colaboração com Alfred N. Witehead, tornou-se a obra de referência da lógica matemática. O esforço dos matemáticos foi o de dar à álgebra uma estrutura lógica, procurando caracterizar a matemática não tanto pelo seu conteúdo quanto pela sua forma.

Fonseca (1998) diz que a partir de 1904, com David Hilbert, inicia-se um novo período dessa ciência então emergente, que se caracteriza pela aparição da metalógica ou metamatemática, e a partir de 1930, por uma sistematização formalista desta mesma metalógica. Iniciaram-se discussões sobre o valor e os limites da axiomatização, o nexo entre lógica e matemática.

Após estas contribuições decisivas, os lógicos acabaram por se dividir quanto às relações entre a lógica e a matemática, tendo surgido três escolas, segundo Fonseca (1998):

a) os logiscistas, que defendiam que a lógica era um ramo da matemática;
b) os formalistas, que defendiam que ambas as ciências eram independentes, mas formalizadas ao mesmo tempo;
c) os intuicionistas, para os quais a lógica era um derivado da matemática porque era axiomatizada.

2.1.3 Período Contemporâneo

Foi no século XVII que começou uma sucessão de notáveis investigações e invenções que iriam conduzir à inteligência artificial. As idéias filosóficas do tempo estimulavam estas descobertas.

No século XIX, as ligações entre a lógica e a matemática vieram a demonstrar a possibilidade de conceber as operações mentais como simples cálculos, susceptíveis de serem executados por máquinas.

Conforme Fontes (2003), nesta fase a lógica matemática se expande e diversifica-se para outros ramos no estudo da lógica, como a lógica combinatória, lógica modal, lógica polivalente e lógica deôntica, entre outras. A lógica matemática veio a exercer influência decisiva em muitos ramos da computação.

Destacam-se também neste período alguns sistemas originais de matemáticos como Schönfinkel, Curry, Kleene, Rosser e Alonzo Church. Quase todos estes últimos matemáticos, junto com o logiscista inglês Alan M. Turing, acabaram por definir, antes mesmo de existir o computador propriamente, a natureza da computação e as implicações e limites do pensamento humano através de uma máquina. (FONSECA, 1998).

Segundo Abe (2002), com a evolução dos conceitos da lógica, no século XX, o seu estudo atingiu um elevado grau de formalização, tornando, assim, a lógica adapta a ser aplicada à concepção de máquinas inteligentes. Contudo, o desenvolvimento dos computadores acabou por impulsionar o aparecimento da lógica para ciências da computação.

2.2 ÁREAS DE APLICAÇÃO

Segundo Nahra (1997, p. 121) a lógica “é uma ciência que pode ser aplicada em várias outras ciências e em vários ramos do conhecimento humano”.

Por exemplo, a lógica é importante para as funções desenvolvidas pelo psicólogo, pois através da análise lógica dos argumentos, pode-se descobrir as inferências sobre os princípios que movem as ações das pessoas, permitindo a descoberta de traços característicos da personalidade (NAHRA, 1997).

Ainda segundo Nahra (1997), a lógica também tem papel importante nas atividades de um sociólogo, devido ao preparo que se deve ter ao fazer uma pesquisa de opinião, onde se leva em conta à lógica para escolher o tipo de amostra adequada e também para a elaboração do questionário que não pode ser ambíguo.

No direito a lógica pode ser aplicada, por exemplo, por um advogado para fazer uma petição, pois esta precisa ser coerente. Equívocos lógicos facilitam a contestação por adversários.

Existem outras áreas de aplicação da lógica, não só estas citadas. Ela também é aplicada no dia a dia quando alguma decisão precisa ser tomada. A sua aplicação é vasta, novas áreas de aplicação vão surgindo e outras vão se tornando mais claras. Neste trabalho se torna importante apresentar a utilidade da lógica na ciência da computação, como é visto a seguir.

2.3 LÓGICA PARA COMPUTAÇÃO: CÁLCULO PROPOSICIONAL

Callegari (2002) define o cálculo proposicional como:

> Um sistema formal para representação do conhecimento em termos de expressões declarativas que expressam proposições, usando letras e símbolos (os quais representam as proposições) e os conectivos lógicos entre elas. A validade depende do padrão das proposições como pequenas unidades de raciocínio.

O conceito mais fundamental do cálculo proposicional é o conceito de proposição. Uma proposição é uma frase afirmativa, isto é, uma frase que afirma que uma situação é de uma determinada maneira, mas que pode ser verdadeira ou falsa, conforme a situação esteja ou não de acordo com o que a frase afirma.

O estudo da lógica neste trabalho de conclusão de curso segue fundamentalmente três passos básicos:

a) estudo da sintaxe, que considera as regras para escrever fórmulas bem formadas a partir de símbolos proposicionais, de pontuação e dos conectivos proposicionais;
b) estudo da semântica, que considera regras para determinar o significado das fórmulas do cálculo proposicional;
c) estudo de métodos, para verificar fórmulas ou argumentos válidos.

2.3.1 Sintaxe

Conforme Souza (2002), o alfabeto do cálculo proposicional é constituído pelo conjunto dos seguintes símbolos:

a) símbolos de pontuação: ( );
b) símbolos de verdade: *true* e *false*;
c) símbolos proposicionais: P, Q, R, S, P_1, Q_1, R_1, S_1, P_2, Q_2, R_2, S_2, ...;
d) conectivos proposicionais: $\neg$ (negação), $\wedge$ (conjunção), $\vee$ (disjunção), $\rightarrow$ (implicação) e $\leftrightarrow$ (equivalência).

Concatenando os símbolos do alfabeto, forma-se um conjunto de fórmulas do cálculo proposicional. No entanto, não é qualquer concatenação de símbolos que é uma fórmula. Assim, é preciso introduzir o conceito de fórmula bem-formada (fbf), para definir uma fórmula significativa. É necessário para se formar uma fórmula do cálculo proposicional seguir as seguintes regras definidas em Souza (2002):

a) todo símbolo de verdade é uma fórmula;

b) todo símbolo proposicional é uma fórmula;

c) se H e G são fórmulas, então:

($\neg$ H) é uma fórmula: negação;

(H $\wedge$ G) é uma fórmula: conjunção;

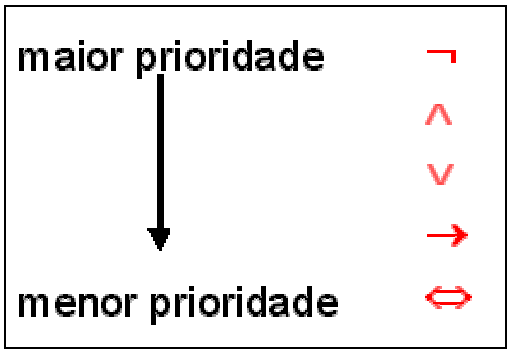
(H $\vee$ G) é uma fórmula: disjunção;

(H $\rightarrow$ G) é uma fórmula: H é o antecedente e G é o conseqüente;

(H $\leftrightarrow$ G) é uma fórmula: H é o lado esquerdo e G é o lado direito.

Cada item citado acima define uma regra para a construção de fórmulas a partir de fórmulas mais simples. Por exemplo, a partir das fórmulas P e Q, obtém-se a fórmula (P $\vee$ Q). Utilizando as fórmulas (P $\vee$ Q) e *true*, obtém-se a fórmula ((P $\vee$ Q) $\rightarrow$ *true*).

No cálculo proposicional, pode ser utilizada uma ordem de precedência dos conectivos para permitir a simplificação das fórmulas com a eliminação de símbolos de pontuação. Embora neste trabalho não seja usada nenhuma ordem de precedência dos conectivos, pois a ordem de precedência dos conectivos varia de autor para autor confundindo assim o aluno, uma ordem de precedência possível para os mesmos é definida na fig. 1.



Fonte: Nunes (2003).

FIGURA 1 - Ordem de precedência dos conectivos

Conforme Souza (2002), a partir das fórmulas da lógica proposicional são definidos alguns elementos sintáticos, como as subfórmulas. Uma subfórmula de uma fórmula H é definida por:

a) H é subfórmula de H;
b) se H = ($\neg$ G), então G é subfórmula de H;
c) se H é subfórmula do tipo (G $\wedge$ E), (G $\vee$ E), (G $\rightarrow$ E) e (G $\leftrightarrow$ E), então G e E são subfórmulas de H;
d) se G é subfórmula de H, então toda subfórmula de G é subfórmula de H.

2.3.2 Semântica

A semântica da lógica proposicional associa a cada objeto sintático um significado. Assim, quando se escreve a fórmula (P $\wedge$ Q), dependendo dos significados de P e Q, esta fórmula pode ser verdadeira (T) ou falsa (F), tendo diferentes significados semânticos.

Conforme Souza (2002), como as fórmulas são formadas pela concatenação de símbolos do alfabeto, sua interpretação é feita a partir da definição da interpretação desses símbolos. A interpretação das fórmulas do cálculo proposicional é feita através de um conjunto de regras semânticas obtidas a partir dos significados semânticos dos símbolos proposicionais, de verdade e dos conectivos proposicionais.

As regras semânticas podem ser representadas por tabelas, denominadas tabelas verdade. Estas tabelas são associadas aos conectivos proposicionais e às fórmulas do cálculo proposicional. Estas tabelas são construídas a partir das tabelas associadas aos conectivos proposicionais, definidas pela tabela 1 a seguir (SOUZA, 2002).

Tabela 1 – Tabela verdade associada a conectivos

P	Q	$\neg$P	P$\vee$Q	P$\wedge$Q	P$\rightarrow$Q	P$\leftrightarrow$Q
T	T	F	T	T	T	T
T	F	F	T	F	F	F
F	T	T	T	F	T	F
F	F	T	F	F	T	T

Diz-se que fórmulas do cálculo proposicional podem, dependendo de suas tabelas verdades, ser classificadas como: tautologias, contraditórias ou satisfatíveis. Estas são as propriedades semânticas fundamentais do cálculo proposicional.

Denomina-se tautologia toda fórmula proposicional que é sempre verdadeira em todas as situações, isto é, a última coluna da sua tabela verdade contém somente valores T. Conforme a tabela abaixo, comprova-se que a fórmula ((P$\wedge$Q)$\rightarrow$(P$\leftrightarrow$Q)) é uma tautologia.

Tabela 2 – Exemplo de tautologia

P	Q	P∧Q	P↔Q	((P∧Q)→(P↔Q))
T	T	T	T	T
T	F	F	F	T
F	T	F	F	T
F	F	F	T	T

Se a fórmula proposicional for sempre falsa em todas as situações possíveis, então ela é denominada contraditória. Neste caso a última coluna da tabela verdade conterá valores F. Construindo a tabela verdade da seguinte fórmula (¬P∧(P∧¬Q)), comprova-se que é uma contradição, conforme verifica-se na tabela a seguir.

Tabela 3 – Exemplo de contradição

P	Q	¬P	¬Q	P∧¬Q	(¬P∧(P∧¬Q))
T	T	F	F	F	F
T	F	F	T	T	F
F	T	T	F	F	F
F	F	T	T	F	F

Mas pode acontecer de uma fórmula proposicional em algumas situações ser verdadeira e em outras falsa. Neste caso, a fórmula dita ser satisfatível. Por exemplo, dada a tabela verdade da fórmula ((P∨Q)→P), pode-se dizer que ela é satisfatível.

Tabela 4 – Exemplo de fórmula satisfatível

P	Q	P∨Q	((P∨Q)→P)
T	T	T	T
T	F	T	T
F	T	T	F
F	F	F	T

2.3.3 Métodos

Para determinar a validade das fórmulas do cálculo proposicional, utiliza-se os métodos da tabela verdade, da árvore semântica e da refutação (negação ou absurdo). Conforme Souza (2002), estes métodos se equivalem em muitos aspectos, mas dependendo da fórmula, há métodos mais eficientes na demonstração da validade.

O método da tabela verdade constitui uma forma relativamente simples de estabelecer o valor de verdade de uma proposição composta, assim como de simplificar as demonstrações lógicas. Entretanto, a tabela verdade não é indicada para fórmulas com muitos símbolos proposicionais, pois o número de linhas da tabela verdade depende do número de símbolos proposicionais que a integram, sendo dado o seguinte teorema por Alencar (1996, p. 29): “a

tabela verdade de uma proposição composta com n proposições simples componentes contém 2^n linhas".

Para construir a tabela verdade de uma fórmula, primeiramente é preciso decompor a fórmula em subfórmulas até o nível atômico. Assim, por exemplo, a seguinte fórmula $((P \rightarrow Q) \rightarrow (\neg P \vee Q))$ pode ser decomposta em subfórmulas conforme tabela abaixo.

Tabela 5 – Primeiro passo para construção da tabela verdade

P	Q	¬P	P→Q	¬P∨Q	((P→Q)→(¬P∨Q))

Em segundo lugar, deve-se atribuir as combinações possíveis de valores T e F para as proposições atômicas (P e Q), conforme tabela 6.

Tabela 6 – Segundo passo para construção da tabela verdade

P	Q	¬P	P→Q	¬P∨Q	((P→Q)→(¬P∨Q))
T	T				
T	F				
F	T				
F	F				

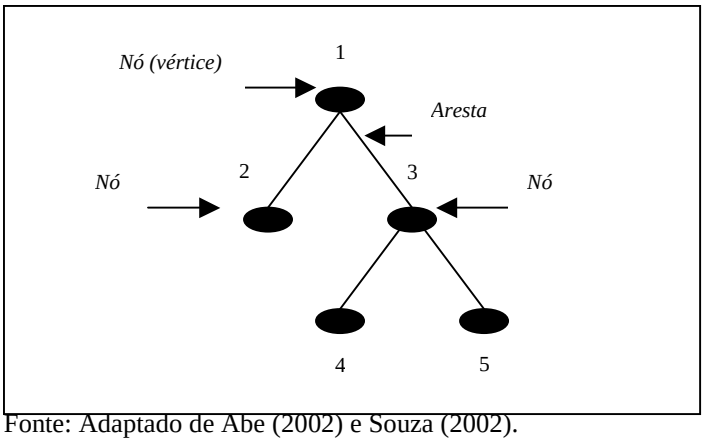
Em seguida, completam-se as colunas escrevendo em cada uma delas os valores lógicos convenientes, no modo abaixo indicado.

Tabela 7 – Terceiro passo para construção da tabela verdade

P	Q	¬P	P→Q	¬P∨Q	((P→Q)→(¬P∨Q))
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

Como na última coluna, onde está a fórmula original, os valores lógicos são todos T, significa então que a fórmula é uma tautologia.

Já o método da árvore semântica determina a validade de uma fórmula a partir de uma estrutura de dados denominada árvore. Uma árvore é um conjunto de nós ou vértices ligados por arestas conforme indicado na fig. 2, onde os nós são rotulados por números inteiros. Os nós finais 2, 4 e 5 são denominados folhas e o nó 1 é a raiz da árvore (SOUZA, 2002).



Fonte: Adaptado de Abe (2002) e Souza (2002).

FIGURA 2 – Exemplo de árvore semântica

Será exemplificada a seguir a validade da fórmula ((P→Q)→(¬P∨Q)) através do método da árvore semântica.

A fig. 3 apresenta a árvore semântica inicial onde as arestas são rotuladas por P=T e P=F, que são as duas possibilidades de valor verdade para P. Na fórmula correspondente ao nó 2, nota-se que é colocado o significado de P. A partir do significado de P, é determinado o significado de ¬P. No caso da fórmula que corresponde o nó 3, apenas com o significado de P é obtido T abaixo do conectivo →, o que significa que a fórmula é verdadeira quando P é falso, independente de Q.

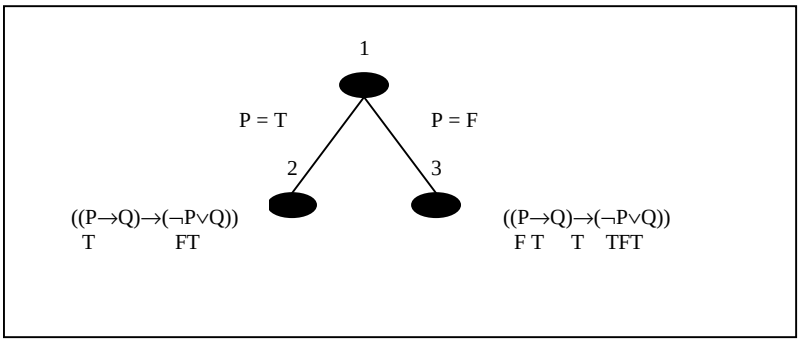


FIGURA 3 – Primeiro passo para construção da árvore semântica

Na fórmula do nó 2 não foi possível deduzir o valor do conectivo →. Então deve-se definir novas arestas que são rotuladas por Q=T e Q=F, conforme fig. 4. Assim, na fórmula do nó 4, todos os significados estão determinados. Um símbolo T abaixo do conectivo → significa que a fórmula é verdadeira quando P e Q são verdadeiros. E seguindo este raciocínio, observa-se que a fórmula do nó 5 também é verdadeira.

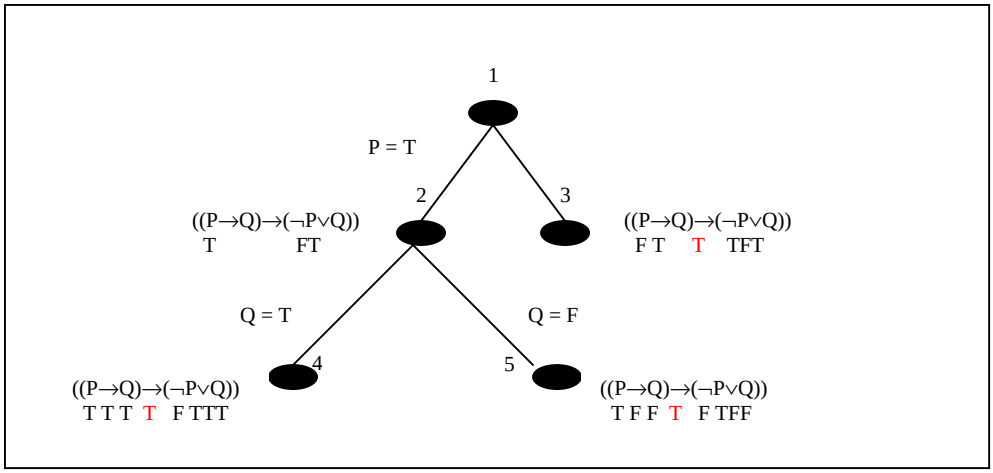


FIGURA 4 – Segundo passo para construção da árvore semântica

Pode-se concluir que a fórmula é uma tautologia, pois o símbolo T aparece em todas as folhas da árvore.

O método da refutação é um método geral de demonstração, utilizado para demonstrar a validade de fórmulas da lógica proposicional. Neste método é considerada inicialmente a negação daquilo que se pretende demonstrar. Por exemplo, considerando uma fórmula H, se o objetivo é demonstrar sua validade, supõe-se que H não é uma tautologia. A partir desta suposição, é utilizado um conjunto de deduções para concluir um fato contraditório ou absurdo. Desta forma, caso se obtenha um absurdo, a conclusão é que a suposição inicial é falsa. Então a não-validade de H é um absurdo. Portanto, a conclusão final é que H é uma tautologia (SOUZA, 2002).

Utilizando a fórmula ((P⟶Q)⟶(¬P∨Q)), pode-se validar esta proposição pelo método da refutação da seguinte forma: inicialmente, traça-se uma tabela com o número de colunas para cada fórmula atômica, e em seqüência nega-se a fórmula em questão, conforme a tabela 8.

Tabela 8 - Primeiro passo do método da refutação

((P	→	Q)	→	(¬	P	∨	Q))
			F				
			1				

Nota-se que o conectivo aplicado o valor F é o da implicação. Como a implicação é F somente quando o antecedente (P⟶Q) é T e o conseqüente (¬P∨Q) é F, pode-se escrever estes valores nas respectivas colunas, obtendo-se a seguinte tabela.

Tabela 9 – Segundo passo do método da refutação

((P	→	Q)	→	(¬	P	∨	Q))
	T		F			F	
	2		1			2	

Aplicando o mesmo raciocínio anterior na fórmula (¬P∨Q), deduz-se que (¬P) é F e Q é F, pois o valor lógico da disjunção de duas proposição é falsa quando P e Q são ambas falsas, conforme tabela abaixo.

Tabela 10 – Terceiro passo do método da refutação

((P	→	Q)	→	(¬	P	∨	Q))
	T		F	F	T	F	F
	2		1	3	4	2	3

Como na fórmula (¬P∨Q) a proposição P é T e Q é F, assume-se na fórmula (P→Q) os mesmos valores respectivamente, conforme tabela abaixo.

Tabela 11 – Quarto passo do método da refutação

((P	→	Q)	→	(¬	P	∨	Q))
T	T	F	F	F	T	F	F
5	2	5	1	3	4	2	3

Observando o resultado na tabela 11, nota-se que a implicação é falsa somente se P é T e Q é F, sendo T nos demais casos. Neste caso, o resultado deveria ser F. Chama-se isto de absurdo. O erro está no primeiro passo ao dizer que a fórmula ((P→Q)→(¬P∨Q)) é F. Portanto, conclui-se que a fórmula em questão é sempre T, logo é uma tautologia.

2.3.4 Argumentos

Segundo Alencar (1996), chama-se de argumento toda a afirmação de que várias proposições (P_1, P_2, ..., P_n) têm por conseqüência uma outra proposição Q. As proposições P_1, P_2, ..., P_n são as premissas do argumento, e a proposição final Q é a conclusão do argumento. O argumento é escrito da seguinte forma $(P_1 \wedge P_2 \wedge ... \wedge P_n) \rightarrow Q$, por exemplo, (P∧(P→Q)∧(Q→R))→R.

Alencar (1996) diz que um argumento é válido se e somente se a conclusão é verdadeira todas as vezes que as premissas são verdadeiras, isto é, um argumento é válido se é uma tautologia. A validade de um argumento depende exclusivamente da relação existente

entre as premissas e a conclusão, pois uma conclusão não pode ser verdadeira quando suas premissas são falsas.

A tabela verdade do argumento $(P \wedge (P \rightarrow Q) \wedge (Q \rightarrow R)) \rightarrow R$ ficaria da seguinte forma.

Tabela 12 – Exemplo de argumento

P	Q	R	P→Q	Q→R	((P →Q)∧(Q→R))	(P∧(P→Q)∧(Q→R))	(P∧(P→Q)∧(Q→R))→R
T	T	T	T	T	T	T	T
T	T	F	T	F	F	F	T
T	F	T	F	T	F	F	T
T	F	F	F	T	F	F	T
F	T	T	T	T	T	F	T
F	T	F	T	F	F	F	T
F	F	T	T	T	T	F	T
F	F	F	T	T	T	F	T

A última coluna desta tabela verdade apresenta somente a letra T. Logo, o argumento em questão é uma tautologia, ou seja, o argumento é válido. Chega-se ao mesmo resultado observando que as premissas (P, P→Q, Q→R) e a conclusão (R) do argumento são verdadeiras (T) na linha 1.

Outro exemplo de argumento é citado na introdução do trabalho, onde é descrito o seguinte problema:

> Você foi convocado a participar do júri em um processo criminal. O advogado de defesa argumenta o seguinte: Se meu cliente fosse culpado, a faca estaria na gaveta. Ou a faca não estava na gaveta ou Jacson Pritchard viu a faca. Se a faca não estava lá no dia 10 de outubro, então Jacson Pritchard não viu a faca. Além disso, se a faca estava lá no dia 10 de outubro, então a faca estava na gaveta e o martelo estava no celeiro. Mas todos sabemos que o martelo não estava no celeiro. Portanto, senhoras e senhores, meu cliente é inocente. (GERSTING, 2001, p. 1)

Para provar a validade desse argumento, é necessário escrevê-lo usando a notação da lógica formal. Deve-se identificar as sentenças atômicas que compõem o argumento, associando símbolos proposicionais a cada uma delas. Assim, tem-se que:

a) P_1: significa “meu cliente é culpado”;
b) P_2: significa “a faca estaria na gaveta”;
c) P_3: significa “Jacson Pritchard viu a faca”;
d) P_4: significa “a faca estava lá no dia 10 de outubro”;
e) P_5: significa “o martelo estava no celeiro”;

O argumento é então escrito da seguinte forma: $((P_1 \rightarrow P_2) \wedge (\neg(\neg P_2 \leftrightarrow P_3)) \wedge (\neg P_4 \rightarrow \neg P_3) \wedge (P_4 \rightarrow (P_2 \wedge P_5)) \wedge \neg P_5) \rightarrow \neg P_1$. E através do método da

refutação, pode-se dizer que o cliente é inocente, ou seja, este argumento é válido, visto que na tabela abaixo existe absurdo em $\neg P_5$ ao tentar provar que o argumento é uma tautologia.

Tabela 13 – Validade do argumento através do método da refutação

$((P_1$	$\rightarrow$	$P_2)$	$\wedge$	$(\neg$	$(\neg$	P_2	$\leftrightarrow$	$P_3))$	$\wedge$	$(\neg$	P_4	$\rightarrow$	$\neg$	$P_3)$	$\wedge$	$(P_4$	$\rightarrow$	$(P_2$	$\wedge$	$P_5))$	$\wedge$	$\neg$	$P_5)$	$\rightarrow$	$\neg$	P_1
T	T	T	T	T	F	T	F	T	T	F	T	T	F	T	T	T	T	T	T	T	T	T	T	F	F	T
4	5	6	2	5	9	7	8	10	2	12	13	5	11	10	2	14	5	7	15	16	2	5	16	1	2	3

3 PROCESSADORES DE LINGUAGENS

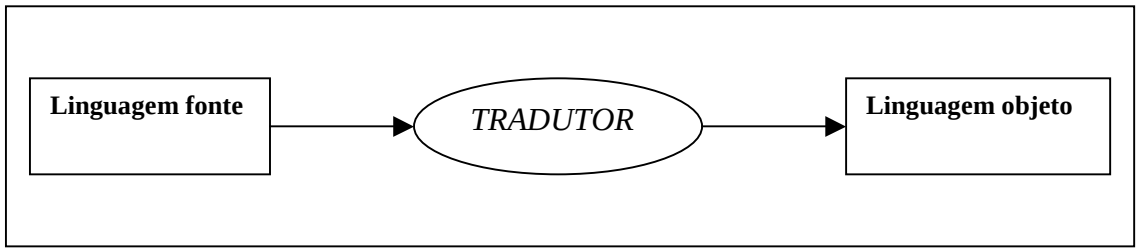
Conforme Price e Toscani (2001, p. 1), na programação de computadores “uma linguagem de programação serve como meio de comunicação entre o indivíduo que deseja resolver determinado problema e o computador escolhido para ajudá-lo na solução”, ou seja, faz a ligação entre homem e máquina.

As primeiras linguagens de programação introduzidas foram as linguagens de baixo nível, são linguagens baseadas em código de máquina e de difícil utilização. Mais tarde, para facilitar a programação, surgiram as linguagens de alto nível, que utilizam sintaxe mais estruturada tornando o código mais fácil de se entender e de editar programas.

Hoje as linguagens de alto nível são as mais utilizadas. Para que se tornem operacionais, é preciso que os programas escritos em linguagens de alto nível sejam traduzidos para linguagem de máquina. Esta conversão é realizada através de tradutores ou processadores de linguagens.

3.1 TIPOS DE TRADUTORES DE LINGUAGENS DE PROGRAMAÇÃO

Segundo Aho (1995), o tradutor é um sistema que lê um programa escrito numa linguagem (linguagem fonte) e o traduz num programa equivalente numa outra linguagem (linguagem objeto), conforme fig. 5.



Fonte: Jose Neto (1987).

FIGURA 5 - Esquema de conversão de um tradutor

Price e Toscani (2001) classificam os tradutores de linguagens de programação em:

a) montadores: são aqueles tradutores em que a linguagem-fonte é de baixo nível. Mapeiam instruções em linguagem simbólica (*assembly*) para instruções de linguagem de máquina, numa relação de uma-para-uma;
b) macro-assemblers: mapeiam instruções em linguagem simbólica (*assembly*) para instruções de linguagem de máquina, numa relação de uma-para-várias;

c) compiladores: convertem programas escritos em linguagem de alto nível para programas equivalentes em linguagem simbólica ou em linguagem de máquina;
d) pré-compiladores: são tradutores que efetuam conversões entre duas linguagens de alto nível;
e) interpretadores: são processadores que aceitam como entrada o código intermediário de um programa anteriormente traduzido e produzem o efeito da execução do algoritmo original sem, porém, mapeá-lo para linguagem de máquina.

Basicamente, a interpretação se caracteriza por realizar as fases de compilação e execução, de cada um dos comandos encontrados no programa. Na realidade o interpretador lê cada linha do programa fonte, extrai os comandos nela presentes, verifica a sua sintaxe, e os executa diretamente. Não existem etapas intermediárias. Caso uma linha de programa seja executada mais de uma vez, ela deverá ser novamente interpretada.

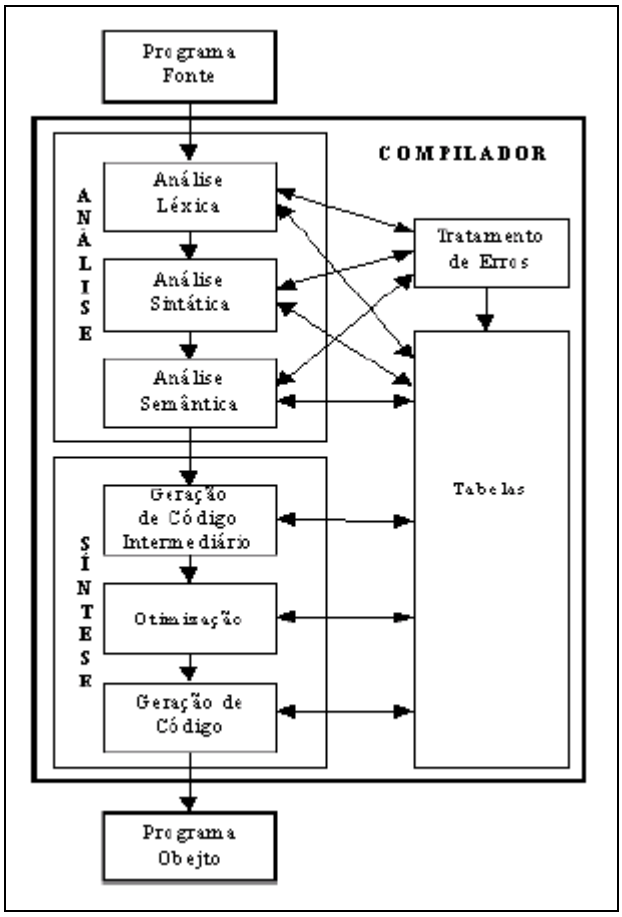
Já os compiladores, quando traduzem um programa escrito em linguagem de alto nível, produzem um programa em linguagem objeto, que uma vez gerado pode ser executado uma ou mais vezes no futuro. Assim, uma vez compilado um programa, enquanto o código fonte do programa não for alterado, ele poderá ser executado sucessivas vezes, sem necessidade de nova compilação.

3.2 ESTRUTURA DE UM TRADUTOR

A construção de um tradutor é um tanto complexa. Contudo, a utilização de técnicas que vem sendo desenvolvidas desde as primeiras implementações, leva a uma estruturação que visa dividir a tarefa da tradução em tarefas de dimensões menores e com funções específicas.

Uma vez que o processo de tradução tem por objetivo principal traduzir um programa-fonte escrito em uma linguagem fonte para um outro programa equivalente em uma linguagem objeto, pode-se dividi-lo em duas grandes fases: análise e síntese. A primeira determina a estrutura e o significado do programa fonte afim de transformá-lo em um formato intermediário, o qual será utilizado pela segunda fase, a síntese. Esta fase tem a tarefa de mapear a estrutura e o significado do programa fonte na linguagem objeto, gerando um programa objeto.

Seguindo essa linha, a fase de análise também pode ser desmembrada em tarefas com funções específicas. São elas: a análise léxica, a análise sintática e a análise semântica. A fase de síntese, por sua vez, constitui-se da geração e otimização de código. Adicionalmente a estas fases, existem ainda duas fases que interagem com todas as fases do compilador: o gerenciamento de tabelas e o tratamento de erros. A fig. 6 ilustra a estrutura de um tradutor, mais especificamente de um compilador.



Fonte: Price e Toscani (2001).

FIGURA 6 – Estrutura de um compilador

A fase de análise léxica tem por objetivo identificar seqüências de caracteres que constituem unidades léxicas (*tokens*). O analisador léxico lê, caractere por caractere, o texto fonte, identificando *tokens*, e desprezando comentários e espaços em brancos desnecessários. Além disso, nesta fase inicia a construção da tabela de símbolos e a emissão de mensagens de erro caso sejam identificadas unidades léxicas não aceitas pela linguagem em questão. A saída do analisador léxico é uma cadeia de *tokens* que é passada para a próxima fase, a análise sintática.

A fase de análise sintática tem por função verificar se a estrutura gramatical do programa está correta, isto é, se essa estrutura foi formada usando as regras gramaticais da linguagem.

A fase de análise semântica tem por função verificar se as estruturas do programa irão fazer sentido durante a execução. É nesta fase que são detectados, por exemplo, os conflitos entre tipos, a ausência de declarações de variáveis, de funções e de procedimentos.

A fase de geração de código intermediário permite a geração de instruções para uma máquina abstrata, normalmente em código de três endereços, mais adequadas à fase de otimização.

A fase de otimização analisa o código no formato intermediário e tenta melhorá-lo de tal forma que venha a resultar um código de máquina mais rápido em tempo de execução. Uma das tarefas executadas pelo otimizador é a detecção e a eliminação de movimento de dados redundantes e a repetição de operações dentro de um mesmo bloco de programa.

E, por fim, a fase de geração de código tem como objetivo analisar o código já otimizado e gerar o código objeto definitivo para uma máquina alvo. Nesta etapa, as localizações de memória são selecionadas para cada uma das variáveis usadas pelo programa.

Como no presente trabalho serão empregados princípios e técnicas de construção de compiladores como base para construir o interpretador de fórmulas do cálculo proposicional, a seguir serão abordadas com mais detalhes as fases implementadas neste trabalho.

3.2.1 Análise Léxica

Segundo Aho (1995), o analisador léxico é a primeira fase de um compilador. Sua tarefa principal é ler os caracteres de entrada e traduzi-los para uma seqüência de símbolos léxicos, também chamados *tokens*. Exemplo de símbolos léxicos são palavras reservadas, identificadores, constantes e operadores da linguagem.

O significado previamente estabelecido de um símbolo léxico está associado a uma classificação determinada na definição da linguagem. Em decorrência da classificação dos símbolos léxicos, outras tarefas ficam sob a incumbência da análise léxica. Uma delas se baseia no fato de que é necessário uniformizar o tratamento dos símbolos léxicos tanto para os

outros componentes do compilador, como para o próprio analisador léxico. Isso é feito associando uma representação numérica a cada classe de símbolo reconhecida, evitando a dificuldade de lidar com seqüências de caracteres de comprimentos variáveis. O analisador sintático poderá ter seu funcionamento mais simplificado se os símbolos, por ele utilizados, já estiverem classificados com uma numeração. Outra função pertinente ao analisador léxico é a eliminação de delimitadores, espaços em branco e comentários (AZEVEDO, 1998).

Dentre as inúmeras classes de símbolos que podem ser determinadas para o reconhecimento léxico, a classe que representa os identificadores deve ter uma atenção especial. Uma vez que toda a linguagem de programação deve possuir identificadores para manipular os valores atribuídos às variáveis, apesar de pertencerem à mesma classe, dois identificadores podem possuir atributos diferentes, o que torna necessário um modo adicional de diferenciação entre eles. A abordagem mais comum é a utilização de uma tabela de símbolos, a qual contém todos os símbolos encontrados no reconhecimento.

Segundo Price e Toscani (2001), a tabela de símbolos é uma estrutura de dados usada para associar um símbolo léxico, geralmente os identificadores, aos seus atributos, tais como tipo, endereço, conteúdo, dimensão, linha declarada e tudo mais que seja necessário utilizar em fases posteriores do processo de compilação. Cabe ao analisador léxico iniciar a tarefa de preencher a tabela de símbolos com identificadores, ou outros tipos de símbolos léxicos. Feito isso, os identificadores podem ser diferenciados entre si, pela sua posição na tabela.

Além de identificar os símbolos léxicos, o analisador realiza outra função que é emitir mensagens de erro caso identifique unidades léxicas não aceitas pela linguagem em questão.

E, por fim, o resultado do analisador léxico é uma cadeia de *tokens*, que são especificados por expressões regulares.

3.2.1.1 Especificação dos *tokens*: expressões e definições regulares

Os *tokens* são padrões que podem ser especificados através de expressões regulares. A expressão regular é uma ferramenta necessária para o reconhecimento de padrões, sendo uma notação bastante clara e concisa para especificar *tokens*. Para exemplificar uma expressão regular, será especificado o símbolo proposicional, que é composto por uma letra maiúscula

(P, Q, R, S) seguida por zero ou mais dígitos (definido pelo operador *), conforme quadro abaixo.

```
<símbolo proposicional>::= <letra> <digito>*
```

QUADRO 1 – Exemplo de expressão regular

Segundo Aho (1995), uma expressão regular é constituída de expressões regulares mais simples usando-se um conjunto de definições regulares. As definições regulares atuam como expressões auxiliares. Para o exemplo em questão, as definições regulares são representadas no quadro abaixo.

```
<dígito> ::= 0 | 1| ... | 9
<letra>  ::= P | Q | R | S
```

QUADRO 2 – Exemplo de definições regulares

3.2.2 Análise Sintática

Segundo Price e Toscani (2001), a função central de um analisador sintático é verificar se as construções usadas no programa estão gramaticalmente corretas.

O analisador sintático deve avaliar o conjunto de *tokens* identificados pelo analisador léxico, tentando construir uma árvore de derivação[1] (ou árvore de análise sintática) válida, conforme a seqüência dos *tokens* lidos, caso contrário, emite uma mensagem de erro. Esta árvore de derivação deve obedecer às regras de produção de uma gramática livre de contexto, popularizada pela notação *Backus Naur Form* (BNF).

3.2.2.1 Especificação das regras sintáticas

Conforme Jose Neto (1987, p. 117), a notação BNF “trata-se de uma notação recursiva de formalização da sintaxe de linguagens através de produções gramaticais, permitindo assim a criação de dispositivos de geração de sentenças”.

A BNF usa abstrações para representar estruturas sintáticas. As abstrações na descrição BNF são chamadas de símbolos não-terminais, ou simplesmente não-terminais. A

[1] “Árvores de derivação são formas bidimensionais de representação da análise ou de síntese de uma cadeia, com base em uma gramática” (JOSE NETO, 1987, p. 57).

representação para os nomes das abstrações (não-terminais) em BNF é um nome cercado pelos símbolos de menor e maior (< e >). Os *tokens* da linguagem são chamados terminais.

Segundo Jose Neto (1987), uma regra BNF tem sempre um não-terminal em seu lado esquerdo e um conjunto composto por terminais e/ou não-terminais em seu lado direito. O símbolo ::= é usado com o sentido de “é definido por” e une o lado esquerdo ao direito da regra. O símbolo | é usado com o significado de “ou” e é usado para não precisar escrever o lado esquerdo repetidas vezes. A seguir uma especificação inicial das regras sintáticas das fórmulas do cálculo proposicional.

```
<fórmula> ::=   <símbolo proposicional>
              | <símbolo verdade>
              | (<fórmula> <conectivo> <fórmula>)
              | ¬ <fórmula>
<conectivo>::= ∨ | ∧ | → | ↔
<símbolo verdade> ::= T | F
```

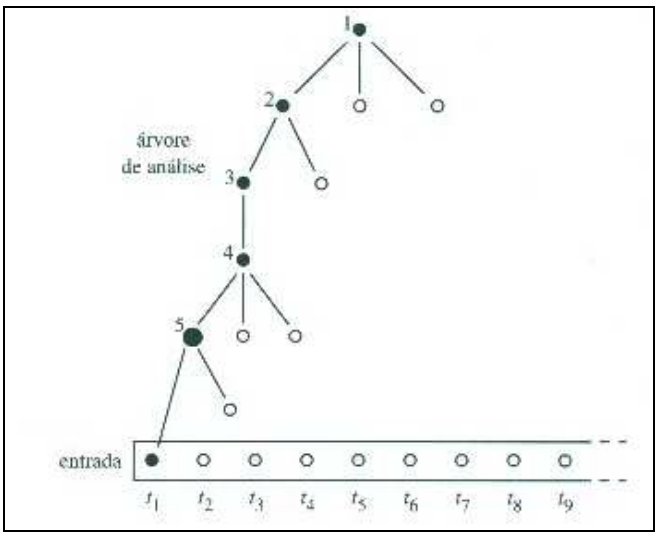
QUADRO 3 – BNF do cálculo proposicional

3.2.2.2 Tipos de análise sintática

Há duas estratégias básicas para a análise sintática:

a) *top-down* (descendente): constrói a árvore de derivação a partir do topo (raiz), fazendo a árvore crescer até atingir suas folhas;
b) *bottom-up* (redutiva): realiza a análise no sentido inverso, começa pelas folhas e constrói a árvore até a raiz.

Segundo Grune (2002), um analisador sintático *top-down* pode ser visto como uma tentativa de construir uma árvore de derivação em pré-ordem, o que significa que a parte superior de uma subárvore é construída antes de qualquer de seus nós, conforme figura abaixo.



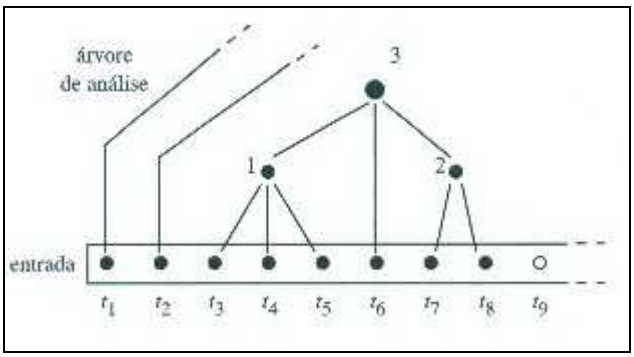
Fonte: Grune(2002).

FIGURA 7 – Analisador *top-down*: árvore de análise sintática

Os principais analisadores sintáticos *top-down* são:

a) recursivo descendente: este analisador é construído como um conjunto de rotinas representando cada não-terminal da gramática. Cada símbolo na cadeia de entrada fará com que o analisador decida entre chamar uma nova produção da gramática ou reconhecer o terminal na cadeia de entrada;
b) LL(1): significa que as sentenças geradas pela gramática são passíveis de serem analisadas da esquerda para a direita (L=*left-to-right*), produzindo uma derivação mais a esquerda (L=*leftmost*), levando em conta apenas um (1) símbolo da entrada.

Grune (2002) diz que o método de análise *bottom-up* constrói os nós da árvore de derivação em pós-ordem, ou seja, a parte superior de uma subárvore é construída após todos os seus nós inferiores terem sido construídos, como é visto na fig. 8.



Fonte: Grune (2002).

FIGURA 8 – Analisador *bottom-up*: árvore de análise sintática

Conforme Price e Toscani (2001), a análise *bottom-up* é também denominada redutiva, pois a sentença de entrada é reduzida até atingir o símbolo inicial da gramática (raiz da árvore de derivação).

Um dos principais tipos de analisadores *bottom-up* é o LR(k). Os analisadores sintáticos LR(k) lêem a sentença em análise da esquerda para a direita (L=*left-to-rigth*) e produzem uma derivação mais à direita ao reverso (R=*rightmost derivation*), considerando k símbolos entrada.

Segundo Price e Toscani(2001), há basicamente três tipos de analisadores LR:

a) SLR(*Simple* LR), fáceis de implementar, porém aplicáveis a uma classe restrita de gramáticas;
b) LR canônicos, mais poderosos, podendo ser aplicados a um grande número de gramáticas;
c) LALR (*Look Ahead* LR), de nível intermediário e implementação eficiente, que funciona para a maioria das linguagens de programação.

3.2.3 Análise Semântica

A principal função da análise semântica é determinar se as estruturas sintáticas analisadas fazem sentido, ou seja, verificar se o programa não possui erros de significado. Um importante componente da análise semântica é a verificação de tipos. Nela o compilador checa se cada operador recebe os operandos que são permitidos pela especificação da linguagem fonte. Por exemplo, conforme Price e Toscani (2001), em Pascal, o comando while tem a sintaxe apresentada na fig. 9.

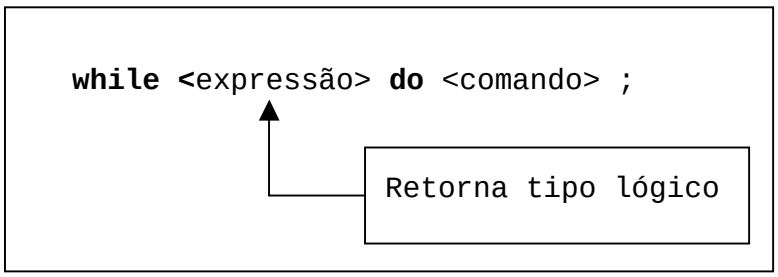


FIGURA 9 – Forma geral do comando while

A estrutura <expressão> deve apresentar-se sintaticamente correta e sua avaliação deve retornar um valor do tipo lógico. Isto é, a aplicação de operadores sobre os operandos deve resultar num valor do tipo lógico.

3.2.4 Geração de Código Intermediário

Conforme Price e Toscani (2001), esta fase utiliza a representação interna produzida pelo analisador sintático e gera como saída uma seqüência de código. Esse código pode ser o código objeto final, mas, na maioria das vezes, constitui-se num código intermediário. A grande diferença entre estes dois códigos é que o intermediário não especifica detalhes da máquina alvo, tais como quais registradores serão usados, quais endereços de memória serão referenciados.

Os vários tipos de código intermediário fazem parte de uma das seguintes categorias, segundo Price e Toscani (2001):

a) representações gráficas tais como árvores de sintaxe abstratas ou grafos de sintaxe;
b) notação pós-fixada ou pré-fixada;
c) código de três-endereços (triplas e quádruplas).

Uma árvore de sintaxe abstrata é uma forma condensada de árvore de derivação na qual somente os operandos da linguagem aparecem como folhas e os operadores constituem nós interiores da árvore, conforme apresentado na fig. 10. Já um grafo de sintaxe, além de incluir as simplificações da árvore de sintaxe, faz a fatoração das subexpressões eliminando-as.

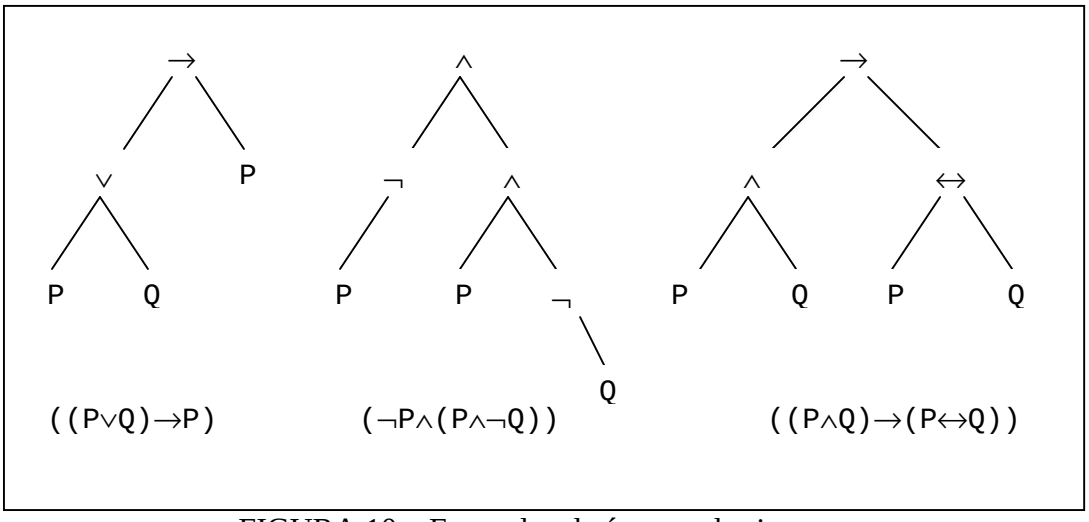


FIGURA 10 – Exemplos de árvores de sintaxe

A notação tradicional para expressões aritméticas, que representa uma operação binária na forma x+y, ou seja, com o operador entre seus dois operandos, é conhecida como notação infixada. Uma notação alternativa para esse tipo de expressão é a notação pós-fixada, também

conhecida como notação polonesa, na qual o operador é expresso após seus operandos. Ou então a notação pré-fixada, onde primeiro é expresso o operador e depois os operandos, da esquerda para a direita. Na tabela 14 são apresentadas fórmulas do cálculo proposicional nas notações infixada, pós-fixada e pré-fixada, respectivamente.

Tabela 14 – Notações infixadas, pós e pré fixadas

NOTAÇÃO		
infixada	***pós-fixada***	***pré-fixada***
((P∨Q)→P)	PQ∨P→	→∨PQP
(¬P∧(P∧¬Q))	P¬PQ¬∧∧	∧¬P∧P¬Q
((P∧Q)→(P↔Q))	PQ∧PQ↔→	→∧PQ↔PQ

O código de três endereços é composto por uma seqüência de instruções envolvendo operações binárias ou unárias e uma atribuição. O nome “três endereços” está associado à especificação, em uma instrução, de no máximo três variáveis: duas para os operadores binários e uma para o resultado. Assim, expressões envolvendo diversas operações são decompostas nesse código em uma série de instruções, eventualmente com a utilização de variáveis temporárias introduzidas na tradução. Dessa forma, obtém-se um código mais próximo da estrutura da linguagem *assembly* e, conseqüentemente, de mais fácil conversão para a linguagem-alvo.

Conforme Aho (1995), a representação interna das instruções em códigos de três endereços dá-se na forma de armazenamento em tabelas com quatro ou três colunas. Na abordagem que utiliza quádruplas (as tabelas com quatro colunas), cada instrução é representada por uma linha na tabela com a especificação do operador, do primeiro argumento, do segundo argumento e do resultado. A tabela abaixo mostra a representação do comando A:= B * (- C + D) em quádruplas.

Tabela 15 – Representação em quádruplas

	OPER	**ARG1**	**ARG2**	**RESULT**
(0)	-	C		T1
(1)	+	T1	D	T2
(2)	*	B	T2	T3
(3)	:=	T3		A

Fonte: Price e Toscani (2001).

A outra forma de representação, por triplas, evita a necessidade de manter nomes de variáveis temporárias ao fazer referência às linhas da própria tabela no lugar dos argumentos. Nesse caso, apenas três colunas são necessárias, uma vez que o resultado está sempre

implicitamente associado à linha da tabela (AHO, 1995). A representação do mesmo comando através de triplas é mostrada abaixo.

Tabela 16 – Representação em triplas

	OPER	ARG1	ARG2
(0)	-	C	
(1)	+	(0)	D
(2)	*	B	(1)
(3)	:=	A	(2)

Fonte: Price e Toscani (2001).

4 DESENVOLVIMENTO DO INTERPRETADOR DE FÓRMULAS DO CÁLCULO PROPOSICIONAL

A partir dos assuntos abordados no capítulo anterior, pôde-se obter o conhecimento necessário para desenvolver o interpretador das fórmulas do cálculo proposicional. Procurou-se desenvolver a ferramenta proposta utilizando o desenvolvimento rápido de aplicações (RAD), descrito em Thiry (2001), que cita os seguintes passos:

a) analisar os requisitos: determinou-se, junto com o cliente, no caso a professora orientadora desse trabalho, as funções que o interpretador deveria possuir;
b) desenvolver um projeto inicial: foi estudado como seria desenvolvido o protótipo;
c) desenvolver dentro de um determinado tempo uma versão da aplicação: foi gerado uma primeira versão do protótipo;
d) entregar a versão para o cliente testar: o cliente testou a versão com exercícios dados em sala de aula;
e) receber *feedback*: o cliente indicou se o protótipo estava de acordo com o que foi solicitado;
f) caso a versão apresente problemas, planejar uma versão para resolvê-los, respondendo a este *feedback*.

4.1 ESPECIFICAÇÃO

Inicialmente será apresentada a especificação da sintaxe da linguagem do cálculo proposicional e, em seguida, a especificação da aplicação.

4.1.1 Especificação da Linguagem do Cálculo Proposicional

Para especificar a sintaxe da linguagem do cálculo proposicional, primeiramente especificou-se os *tokens* (símbolo proposicional, símbolo verdade e conectivos) utilizando definições regulares, conforme quadro abaixo.

```
<dígito> ::= 0 | 1| ... | 9
<letra>  ::= P | Q | R | S
<símbolo proposicional>::= <letra> <digito>*
<conectivo>::= ~ | | | & | -> | <->
<símbolo verdade> ::= T | F
```

QUADRO 4 – Definições regulares de fórmulas do cálculo proposicional

O símbolo proposicional, como já foi visto na seção 3.2.1.1, inicia com uma letra maiúscula (P, Q, R, S) seguida por zero ou mais dígitos (definido pelo operador *), sendo que letra e dígito só são utilizados na definição do símbolo proposicional. O símbolo verdade pode ser T (*true*) ou F (*false*). Observa-se que entre a especificação feita e os conectivos apresentados no capítulo 2, existe a seguinte correspondência: uso de **~** para a negação (¬), uso de **|** para a disjunção (∨), uso de **&** para a conjunção (∧), uso de **->** para a implicação (→) e uso de **<->** para a equivalência (↔).

No quadro 5 é mostrada a especificação da BNF do cálculo proposicional. Tem-se argumentos e fórmulas.

```
<cálculo proposicional> ::= <argumento> | <fórmula>

<argumento> ::= ( <premissas> ) -> <conclusão>
<premissas> ::= <fórmula> | <fórmula> & <premissas>
<conclusão> ::= <fórmula>

<fórmula> ::=   <símbolo proposicional>
              | <símbolo verdade>
              | ~ <fórmula>
              | ( <fórmula> <conectivo> <fórmula> )
```

QUADRO 5 - Especificação da BNF do cálculo proposicional

Conforme definido no capítulo 2, um argumento possui premissas e uma conclusão. As premissas devem estar entre parênteses, podendo ser uma ou mais fórmulas, separadas umas das outras pelo conectivo **&**. A conclusão é precedida pelo conectivo -> e também é uma fórmula do cálculo proposicional.

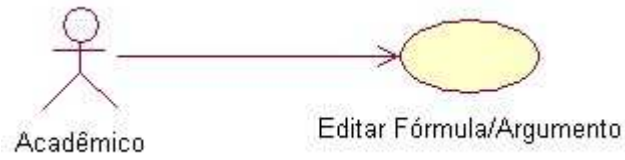
Uma fórmula pode ser um <símbolo proposicional>, um <símbolo verdade>, o conectivo **~** seguido de uma <fórmula> ou uma <fórmula> seguida de um <conectivo> seguido de outra <fórmula> entre parênteses.

A semântica das fórmulas e argumentos é aquela apresentada no capítulo 2.

4.1.2 Especificação da Aplicação

Para a especificação da aplicação foi utilizado a linguagem de modelagem UML (BOOCH, 2000). São apresentados os casos de uso, as principais classes modeladas e o diagrama de seqüência, desenvolvidos com auxílio da ferramenta *Rational Rose* (QUATRANI, 2001).

No quadro 6 tem-se o primeiro caso de uso. A notação para caso de uso utilizada na especificação é uma adaptação daquela apresentada por Thiry (2001).



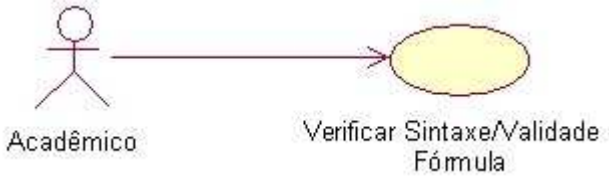
CASO DE USO:	Editar Fórmula/Argumento
BREVE DESCRIÇÃO	No campo `Fórmula/Argumento`, o acadêmico irá digitar uma fórmula ou um argumento do cálculo proposicional.
ATOR(ES):	Acadêmico
PRÉ-CONDIÇÕES:	O acadêmico deve conhecer a notação definida para escrever fórmulas e argumentos do cálculo proposicional no interpretador
FLUXO PRINCIPAL:	O acadêmico irá digitar no campo `Fórmula/Argumento` a fórmula ou argumento do cálculo proposicional.
PÓS-CONDIÇÕES:	
DETALHES DE INTERFACE:	TE-001

QUADRO 6 – Caso de uso: Editar Fórmula/Argumento

Na fig. 11, é apresentada a tela TE-001 que contém o campo `Fórmula/Argumento`, onde a fórmula ou o argumento do cálculo proposicional deve ser digitado. Acima deste campo, encontram-se quatro botões, sendo que o segundo e o terceiro têm por finalidade verificar a sintaxe e validade das fórmulas e dos argumentos, respectivamente. No campo `Resultado` será apresentada a validade da fórmula ou do argumento e no campo `Erro` serão apresentadas mensagens de possíveis erros detectados.

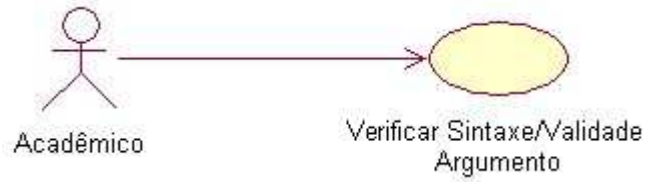
FIGURA 11 – Interface do interpretador de fórmulas do cálculo proposicional

No quadro 7 é apresentado o segundo caso de uso.

	
CASO DE USO:	Verificar Sintaxe/Validade Fórmula
BREVE DESCRIÇÃO	O interpretador verificará a sintaxe e validade da fórmula, retornando no campo `Resultado` a propriedade semântica da mesma.
ATOR(ES):	Acadêmico
PRÉ-CONDIÇÕES:	Uma fórmula deve ter sido digitada no campo `Fórmula/Argumento`.
FLUXO PRINCIPAL:	O acadêmico deverá aplicar o botão para verificar sintaxe/validade da fórmula. O interpretador irá verificar se a fórmula digitada está sintaticamente correta, isto é, se está escrita de acordo com a BNF especificada na seção 4.1.1. Caso seja detectado algum erro sintático, o mesmo será apresentado no campo `Erro`. O interpretador irá determinar a propriedade semântica da fórmula digitada. Ou seja, será apresentado no campo `Resultado` se a fórmula é tautologia, contraditória ou satisfatível.
PÓS-CONDIÇÕES:	
DETALHES DE INTERFACE:	TE-001

QUADRO 7 – Caso de uso: Verificar Sintaxe/Validade Fórmula

No quadro 8 é apresentado o terceiro caso de uso.

	
CASO DE USO:	Verificar Sintaxe/Validade Argumento
BREVE DESCRIÇÃO	O interpretador verificará a sintaxe e validade do argumento, indicando no campo `Resultado` se o argumento é válido ou inválido.
ATOR(ES):	Acadêmico
PRÉ-CONDIÇÕES:	Um argumento deve ter sido digitado no campo `Fórmula/Argumento`.
FLUXO PRINCIPAL:	O acadêmico deverá aplicar o botão para verificar sintaxe/validade do argumento. O interpretador irá verificar se o argumento digitado está sintaticamente correto, isto é, se está escrito de acordo com a BNF especificada na seção 4.1.1. Caso seja detectado algum erro sintático, o mesmo será apresentado no campo `Erro`.

	O interpretador irá determinar se o argumento digitado é válido ou inválido, apresentado o resultado no campo `Resultado`. O argumento será válido somente quando for tautologia.
PÓS-CONDIÇÕES:	
DETALHES DE INTERFACE:	TE-001

QUADRO 8 – Caso de uso: Verificar Sintaxe/Validade Argumento

As classes da aplicação encontram-se representadas no diagrama de classe da fig. 12.

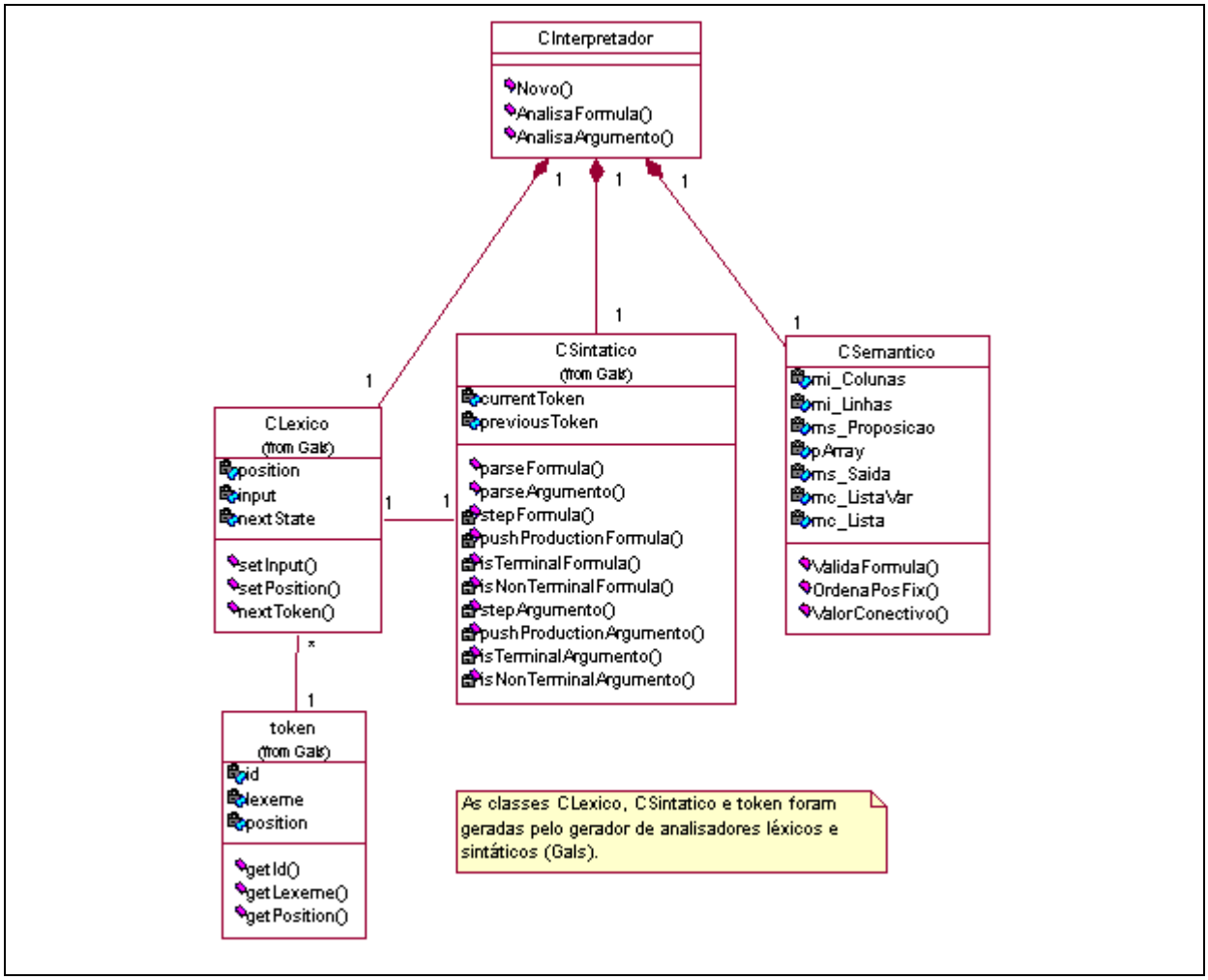


FIGURA 12 – Diagrama de classes

O interpretador de fórmulas do cálculo proposicional foi modelado como uma classe (`CInterpretador`), que possui três métodos: `Novo` para limpar todos os campos, `AnalisaFormula` para verificar a sintaxe e validade das fórmulas, e `AnalisaArgumento` para verificar a sintaxe e validade dos argumentos. Além disso, `CInterpretador` é composto pelas classes `CLexico`, `CSintatico` e `CSemantico`.

A partir da classe `CLexico` é possível verificar a validade dos *tokens*. Seus principais métodos são: `setInput` que passa como entrada a fórmula ou o argumento digitado para o

analisador léxico; `setPosition` que indica a posição a partir da qual o próximo *token* deve ser procurado; `nextToken` que é chamado para se obter o próximo *token* da entrada, sendo dois os resultados possíveis. Caso um *token* seja encontrado, é retornado um novo objeto da classe *token*. Caso nenhum *token* seja reconhecido, ocorrerá uma exceção.

Na classe `CSintatico` será feita a verificação da sintaxe da fórmula ou do argumento usando o método `parseFormula`, no caso de uma fórmula, e `parseArgumento`, no caso de um argumento. Se algum erro for detectado, o método é finalizado ocorrendo uma exceção. Os demais métodos desta classe são usados para implementar o algoritmo LL(1).

Na classe `CSemantico` foram definidos três métodos: `OrdenaPosFix` que irá transformar uma fórmula ou argumento da notação infixada para a notação pós-fixada, `ValidaFormula` que irá verificar a validade das fórmulas e argumentos, e `ValorConectivo` que irá informar o valor lógico da fórmula para cada conectivo encontrado.

Em seguida, será apresentado somente um diagrama de seqüência (fig. 13), o da verificação de sintaxe e validade da fórmula, sendo que o diagrama de seqüência da verificação de sintaxe e validade do argumento é semelhante a este.

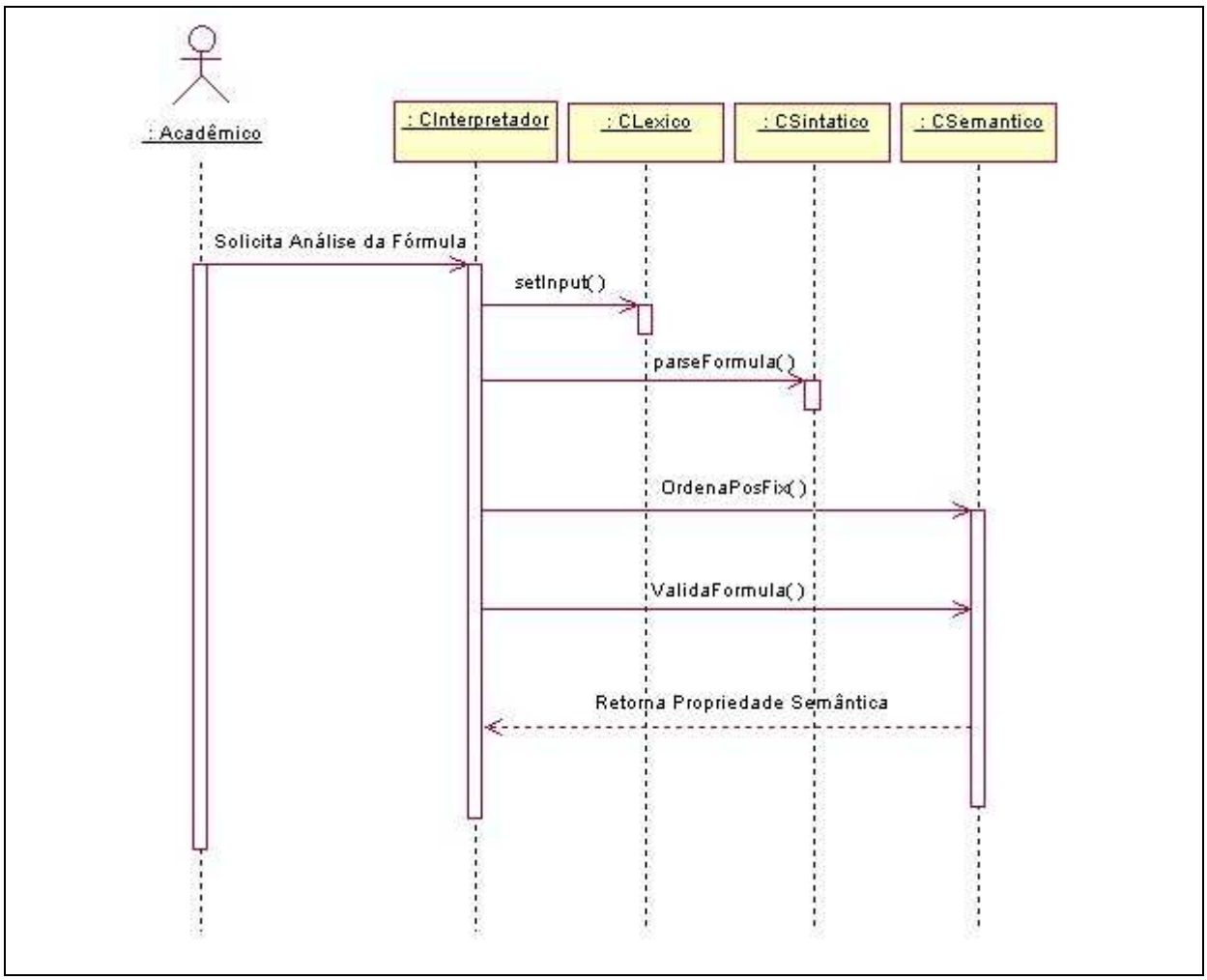


FIGURA 13 – Diagrama de seqüência

A verificação da sintaxe e validade de uma fórmula é executada sempre que o acadêmico, ao terminar de digitar a fórmula, solicitar a verificação da sintaxe e validação da mesma. É instanciado um objeto da classe `CLexico`. Nesta classe é acionado o método `setInput` para passar a fórmula digitada ao analisador léxico. Em seguida, é chamado o método `parseFormula` da classe `CSintatico` que fará a verificação da sintaxe da fórmula (ver código do método no Apêndice 1). Feita a verificação da sintaxe, se não houver erros na fórmula, ativa-se o método `OrdenaPosFix` da classe `CSemantico`, que irá transformar a fórmula da notação infixada para a notação pós-fixada. Em seguida, aplica-se o método `ValidaFormula` que verificará a validade da fórmula.

4.2 IMPLEMENTAÇÃO

Este protótipo foi desenvolvido na linguagem C++ no ambiente Microsoft Visual C++ (BLASZCZAK, 1999). Também foi utilizando o Gerador de Analisadores Léxicos e

Sintáticos (GALS). Segundo Gesser (2001), GALS é uma ferramenta *freeware* que gera analisadores léxicos e analisadores sintáticos em C++, Java ou Delphi. Esta ferramenta tem a opção de gerar analisadores sintáticos recursivos descendentes LL(1) ou analisadores sintáticos redutivos do tipo SLR(1), LALR(1) ou LR(1) canônico.

Os aspectos léxicos de uma especificação GALS são definidos pela declaração dos *tokens* e pela declaração de definições regulares, enquanto os aspectos sintáticos são compostos pela declaração de símbolos terminais (*tokens*), símbolos não-terminais e regras de produções. A fig. 14 apresenta a especificação da BNF do cálculo proposicional descrita de acordo com a notação aceita pela ferramenta GALS.

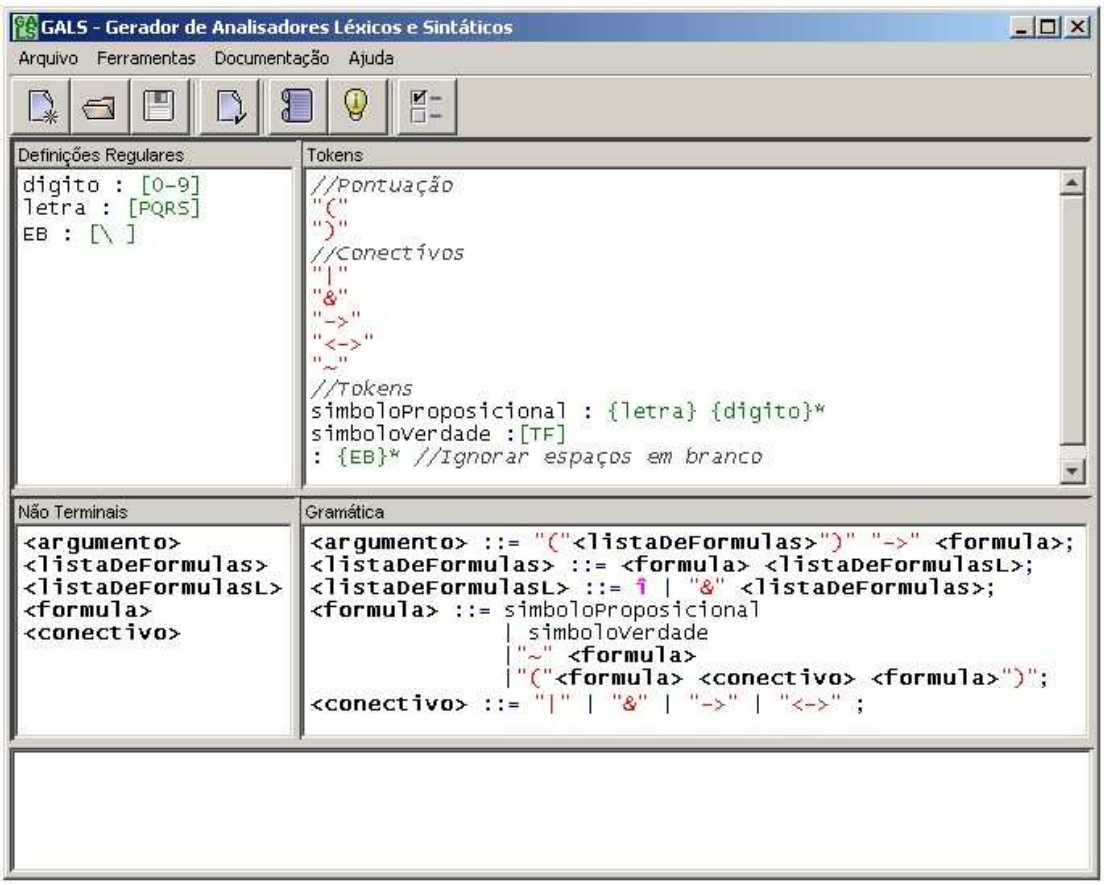


FIGURA 14 - Especificação da BNF do cálculo proposicional no GALS

A partir da especificação mostrada na fig. 14, a ferramenta GALS gerou em C++ o analisador léxico e o analisador sintático recursivo descendente LL(1), que são as classes `CLexico` e `CSintatico` utilizadas na implementação do interpretador de fórmulas do cálculo proposicional.

Para verificar a validade de uma fórmula ou argumento, necessitou-se um algoritmo para gerar a notação pós-fixada. Para sua implementação, seguiram-se os seguintes passos definidos por Moraes (2003):

a) realizar uma varredura da fórmula na sua forma infixada;
b) copiar cada símbolo proposicional encontrado diretamente para a expressão de saída;
c) empilhar cada conectivo ou parênteses de abertura encontrado;
d) quando um parênteses de fechamento for encontrado, desempilhar e copiar símbolos para a expressão de saída até encontrar o parênteses de abertura correspondente.

Em seguida, utilizou-se um algoritmo para avaliar a fórmula na sua forma pós-fixada que segue os seguintes passos definidos por Moraes (2003):

a) realizar uma varredura da fórmula na sua forma pós-fixada;
b) empilhar o valor (T ou F) de cada símbolo proposicional encontrado;
c) quando for encontrado um conectivo, desempilhar os dois últimos valores, exceto para a negação que irá desempilhar apenas um valor, efetuar a operação lógica correspondente e empilhar o resultado novamente na pilha;
d) no final do processo, o resultado da avaliação estará no topo da pilha, sendo armazenado numa matriz;
e) os passos descritos nos itens anteriores são repetidos até que todas as possíveis combinações de valores lógicos para os símbolos proposicionais tenham sido avaliadas. Verifica-se então todas as linhas da última coluna da matriz. Se forem encontrados somente valores T, a fórmula é uma tautologia. Caso sejam obtidos somente valores F, a fórmula é contraditória. E se forem encontrados valores T e F, a fórmula é satisfatível. No caso de um argumento, ao encontrar somente valores T o argumento é dito válido, caso contrário é inválido.

No Apêndice 2 encontra-se a implementação do algoritmo para transformar uma fórmula na notação infixada para pós-fixada e do algoritmo para avaliar a fórmula na sua forma pós-fixada.

4.3 OPERACIONALIDADE DA IMPLEMENTAÇÃO

Como já foi dito anteriormente, com a ferramenta desenvolvida, os acadêmicos da disciplina de Lógica para Computação poderão escrever fórmulas e argumentos do cálculo proposicional, podendo verificar sintaxe e validade dos mesmos.

Assim, para interpretar uma fórmula ou argumento, o acadêmico deverá digitar sua fórmula ou argumento no campo `Fórmula/Argumento`. Há três (3) botões: `novo`, `sintaxe/validade da fórmula` e `sintaxe/validade do argumento`, conforme fig. 15.

FIGURA 15 – Interface do interpretador de fórmulas do cálculo proposicional

Acionando os botões `sintaxe/validade da fórmula` e `sintaxe/validade do argumento`, o acadêmico irá verificar se fórmulas e argumentos foram escritos corretamente. Se ocorrer erro de sintaxe, uma mensagem indicando erro na fórmula ou argumento será visualizada no campo `Erro` e o cursor estará marcando no campo `Fórmula/Argumento` em que posição está o erro, conforme ilustrado na fig 16.

FIGURA 16 – Verificação da sintaxe de uma fórmula

Além da verificação da sintaxe, os botões de verificação `sintaxe/validade da fórmula` e `sintaxe/validade do argumento` também ter por função verificar a validade das fórmulas e argumentos, sendo que se for uma fórmula será apresentada a propriedade semântica da mesma, podendo ser tautologia, contraditória ou satisfatível, através de uma mensagem no campo `Resultado`, conforme fig. 17. No caso de um argumento, este terá como resultado válido ou inválido.

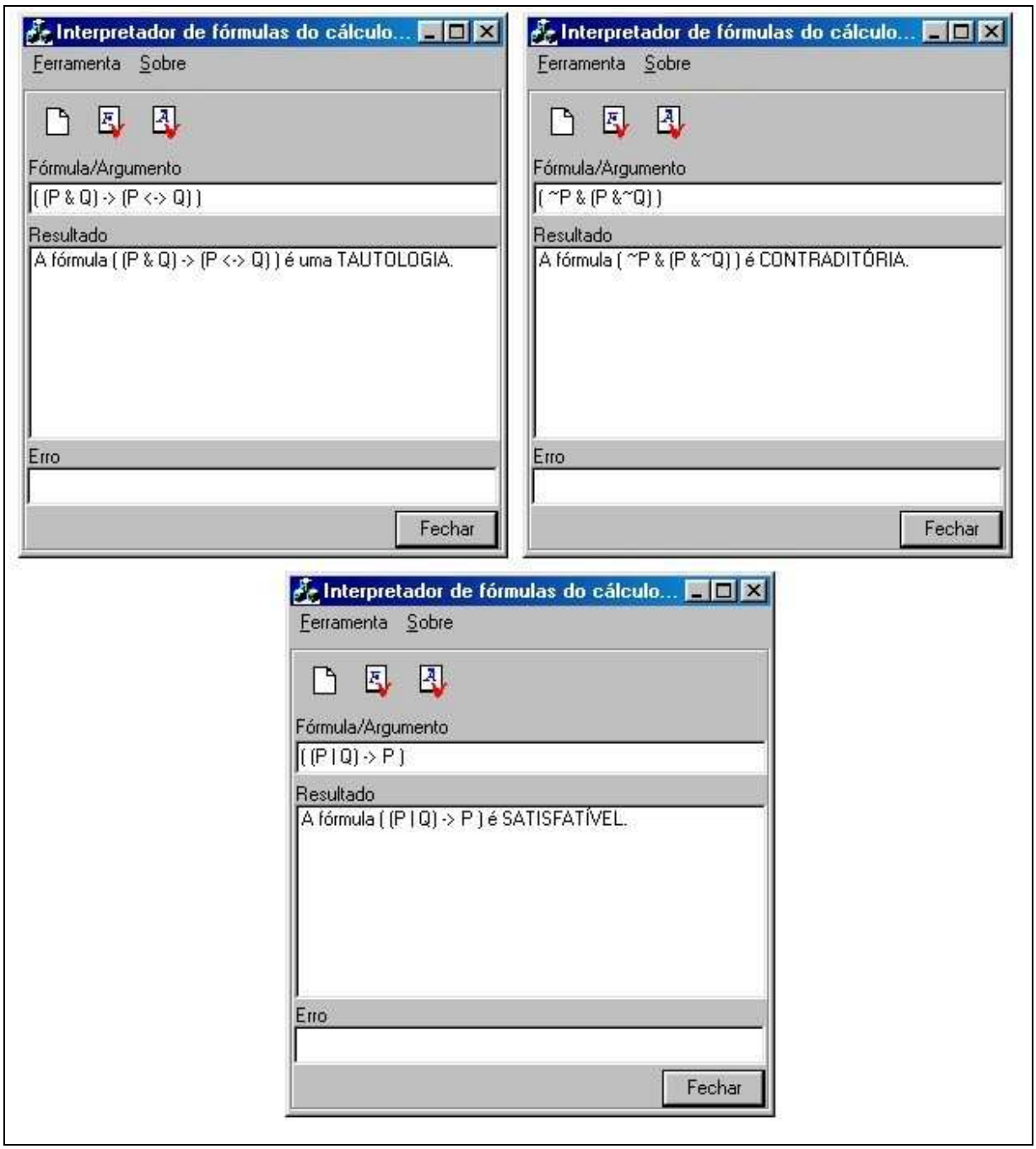


FIGURA 17 – Verificação validade de uma fórmula

5 CONCLUSÕES

O presente trabalho atingiu os objetivos propostos, tendo como resultado final a implementação do interpretador de fórmulas do cálculo proposicional, que permite verificar a sintaxe e a propriedade semântica de uma fórmula, e verificar a sintaxe e a validade de um argumento.

Para este trabalho ser realizado, foram estudados conceitos de lógica e cálculo proposicional para especificar a sintaxe das fórmulas e argumentos, e princípios e técnicas de construção de compiladores como base para construir o interpretador de fórmulas do cálculo proposicional.

Durante o desenvolvimento do interpretador foram utilizadas três ferramentas: *Rational Rose* para fazer a especificação; Microsoft Visual C++ para implementar o interpretador de fórmulas do cálculo proposicional; GALS para gerar os analisadores léxico e sintático utilizados na implementação para verificar a sintaxe das fórmulas e argumentos. O Microsoft Visual C++ é uma ferramenta eficiente para o tipo de programação exigida na implementação do interpretador, porém disponibiliza poucos recursos para a interface. No entanto, pode-se encontrar com facilidade na Internet componentes para suprir esta necessidade. No caso do protótipo desenvolvido, foi utilizado o componente `WBButton` para inserir *bitmaps* nos botões da aplicação. Já o GALS mostrou-se uma ferramenta eficiente e de fácil utilização para geração de analisadores léxicos e sintáticos.

O interpretador de fórmulas do cálculo proposicional é de fácil manipulação e poderá auxiliar professores e acadêmicos da disciplina de Lógica para Computação, porém a possibilidade de se implementar extensões na ferramenta em futuras versões é válida, pois permite o aprimoramento de seu funcionamento abrangendo mais assuntos para facilitar o entendimento da disciplina.

5.1 EXTENSÕES

Pode-se deixar como sugestões para futuros trabalhos as seguintes extensões:

a) possibilitar a visualização da tabela verdade permitindo que o acadêmico verifique os resultados obtidos;
b) incluir outros métodos para determinação da validade das fórmulas do cálculo proposicional, como por exemplo, o método da árvore semântica;
c) implementar métodos de dedução através de regras de inferência e equivalência;
d) possibilitar a verificação de fórmulas do cálculo de predicados;
e) possibilitar a formalização do discurso, permitindo que o acadêmico associe sentenças atômicas com símbolos proposicionais, sendo o discurso automaticamente traduzido para uma fórmula do cálculo proposicional.

REFERÊNCIAS BIBLIOGRÁFICAS

ABE, Jair Minoro; SCALZITTI, Alexandre; SILVA FILHO, João Inácio da. **Introdução à lógica para a ciência da computação.** São Paulo: Arte e Ciência, 2002.

AHO, Alfred V.; SETHI, Ravi; ULLMAN, Jeffrey D. **Compiladores:** princípios, técnicas e ferramentas. Rio de Janeiro: LTC, 1995.

ALENCAR FILHO, Edgard de. **Iniciação a lógica matemática.** São Paulo: Nobel, 1996.

AZEVEDO, Edson Eustáchio. **Proposta de utilização da orientação a objetos na construção de compiladores totalmente parametrizados,** Pará, 1998. Disponível em <http://www.etfpa.br/eustakio>. Acesso: em 19 set. 2003.

BLASZCZAK, Mike. **Professional MFC with Visual C++ 6**. Canada: Wrox Press, 1999.

BOOCH, Grady; RUMBAUGH, James; JACOBSON, Ivar. **UML**: guia do usuário: o mais avançado tutorial sobre Unified Modeling Language (UML). Tradução Fábio Freitas. Rio de Janeiro: Campus, 2000.

CALLEGARI, Daniel. **Introdução à lógica aplicada à computação**, [S.l.], [2003?]. Disponível em: <http://www.inf.pucrs.br/~danielc/>. Acesso em 11 nov. 2003.

FERREIRA, Aurélio B.H. **Novo dicionário da língua portuguesa**. Rio de Janeiro: Nova Fronteira, 1975.

FONSECA FILHO, Cléuzio. **História da computação,** Brasília, ago. 1998. Disponível em: <http://www.cic.unb.br/tutores/hci/hcomp/hcomp.html>. Acesso em: 14 ago. 2003.

FONTES, Carlos. **Navegando na filosofia**, [S.l.], [2003?]. Disponível em: <http://afilosofia.no.sapo.pt/index.html>. Acesso em 14 ago. 2003.

GERSTING, Judith L. **Fundamentos matemáticos para a ciência da computação.** Tradução Valéria de Magalhães Iorio. Rio de Janeiro: LTC, 2001.

GESSER, Carlos Eduardo. **Gerador de Analisadores Léxicos e Sintáticos,** Florianópolis, [2001?]. Disponível em: <http://gals.sourceforge.net/help.html#Intro>. Acesso em: 20 nov. 2003.

GRUNE, Dick et al. **Projeto moderno de compiladores:** implementação e aplicações. Rio de Janeiro: Campus, 2002.

JOSE NETO, João. **Introdução à compilação.** Rio de Janeiro: LTC, 1987.

MENDES, Sueli. **Introdução à lógica**, Rio de Janeiro, mar. 2003. Disponível em: <http://www.api.adm.br/ufrj/logica/links.htm >. Acesso em: 31 jul. 2003.

MORAES, Celso Roberto. **Determinação de escopos em expressões**, São Paulo, maio 2003. Disponível em: <http://www.facens.br/site/alunos/download/estrut_dados>. Acesso em: 28 out. 2003.

MORTARI, Cezar Augusto. **Introdução à lógica.** São Paulo: Ed. da UNESP, 2001.

NAHRA, Cínara. **Através da lógica.** Petrópolis: Vozes, 1997.

NUNES, Maria das Graças Volpe. **Cálculo proposicional**, São Carlos, abr. 2003. Disponível em: <http://www.icmc.sc.usp.br/~gracan/download/sce5832/LógicaCP05.html>. Acesso em: 14 ago. 2003.

PRICE, Ana M.A.; TOSCANI, Simão S. **Implementação de linguagens de programação**: compiladores. Porto Alegre: Sagra-Luzzatto, 2001.

QUATRANI, Terry. **Modelagem visual com Rational Rose 2000 e UML.** Tradução Savannah Hartmann. Rio de Janeiro: Ciência Moderna, 2001.

SOUZA, João Nunes de. **Lógica para ciência da computação**: fundamentos de linguagem, semântica e sistemas de dedução. Rio de Janeiro: Campus, 2002.

THIRY, Marcelo; SALM JR., José. **Processamento de desenvolvimento de software com UML,** [S.l.], [2001?]. Disponível em: <http:www.esp.ufsc.br/disc/procuml/>. Acesso: 20 out. 2003.

APÊNDICE 1

```
void Sintatico::parseFormula(Lexico *scanner)
throw (AnalysisError)
{
    this->scanner = scanner;
    this->semanticAnalyser = semanticAnalyser;

    //Limpa a pilha
    while (! stack.empty())
        stack.pop();

    stack.push(DOLLAR);
    stack.push(START_SYMBOL);

    if (previousToken != 0 && previousToken != currentToken)
        delete previousToken;

    previousToken = 0;

    if (currentToken != 0)
        delete currentToken;

    currentToken = scanner->nextToken();

    while ( ! stepFormula() )
        ;
}

bool Sintatico::stepFormula() throw (AnalysisError)
{
    if (currentToken == 0) //Fim de Sentença
    {
        int pos = 0;

        if (previousToken != 0)
            pos = (previousToken->getPosition()+
                   previousToken->getLexeme().size());
        currentToken = new Token(DOLLAR, "$", pos);
    }

    int a = currentToken->getId();
    int x = stack.top();

    stack.pop();

    if (x == EPSILON)
        return false;
    else if (isTerminalFormula(x))
    {
        if (x == a)
        {
            if (stack.empty())
                return true;
            else
            {
                if (previousToken != 0)
                    delete previousToken;
                previousToken = currentToken;
                currentToken = scanner->nextToken();
                return false;
            }
        }
        else
            throw SyntaticError(PARSER_ERROR[x],
                                currentToken->getPosition());
    }
    else if (isNonTerminalFormula(x))
    {
        if (pushProductionFormula(x, a))
            return false;
        else
```

```
                                currentToken->getPosition());
        }
    }
            throw SyntaticError(PARSER_ERROR[x],

    bool Sintatico::pushProductionFormula(int topStack, int tokenInput)
    {
        int p = PARSER_TABLE[topStack-FIRST_NON_TERMINAL][tokenInput-1];
        if (p >= 0)
        {
            int *production = PRODUCTIONS[p];

            //empilha a produção em ordem reversa
            int length = production[0];
            for (int i=length; i>=1; i--)
                stack.push( production[i] );
            return true;
        }
        else
            return false;
    }
```

APÊNDICE 2

```
void CSemantico::OrdenaPosFix(CString Formula)
{
	std::stack<char> simbolos;
	simbolos.empty();

	CString Infix = "";
	mc_ListaVar.RemoveAll();
	bool bNegacao = false;
	int  iNegacao = 0;
	int iLin = 0;
	int iCol = 0;
	char Tipo = ' ';

	mi_Colunas = 0;
	ms_Saida = "";

	//Obtém-se a fórmula na sua forma infixada
	//excluindo os espaços em branco
	int iTam = Formula.GetLength();
	for (int a = 0; a < iTam; a++ )
	{
		if (Formula.GetAt(a) == ' ')
			continue;
		else
			Infix += Formula.GetAt(a);
	}

	//Varre-se a fórmula
	//Caso for um conectivo ou um abre parênteses coloca na pilha
	//Se for uma letra copia para a saída
	//Ao encontrar um fecha parênteses desempilhar a pilha e copiar
	 //Para a saída até encontrar um abre parênteses
	int iTamanho = Infix.GetLength();
	for (int i= 0; i< iTamanho; i++ )
	{
		Tipo = Infix.GetAt(i);

		if( Tipo == '-' )
			i++;
		else
			if( Tipo == '<' )
				i = 2 + i;
			switch (Tipo)
			{
			case ' ':
				break;
			case '(':
				simbolos.push(Tipo);
				break;
			case ')':
				{
					Tipo = ' ';
					if(simbolos.size() != 0)
						Tipo = simbolos.top();

					if (Tipo != '(')
						ms_Saida += Tipo;

					if(simbolos.size() != 0)
						simbolos.pop( );

					if(simbolos.size() != 0)
					{
						Tipo = simbolos.top();
						simbolos.pop( );
					}

					while (Tipo != '(')
					{
```

```
                        if(simbolos.size() != 0)
                        {
                        ms_Saida += Tipo;
                                Tipo = simbolos.top();
                                simbolos.pop( );
                                ms_Saida += Tipo;
                        }
                        if(simbolos.size() != 0)
                                Tipo = simbolos.top();
                }
        }
        break;
case '|':
        simbolos.push(Tipo);
        mi_Colunas++;
        break;
case '&':
        simbolos.push(Tipo);
        mi_Colunas++;
        break;
case '-':
        simbolos.push(Tipo);
        mi_Colunas++;
        break;
case '<':
        simbolos.push(Tipo);
        mi_Colunas++;
        break;
case '~':
        {
                if (i == 0)
                {
                        if (Infix.GetAt(i+1) == '(')
                                bNegacao = true;
                }
                else
                        if (Infix.GetAt(i+1) == '(')
                                simbolos.push(Tipo);

                if (Infix.GetAt(i+1) == '~')
                        iNegacao++;
                mi_Colunas++;
        }
        break;
default:
        ms_Saida += Tipo;
        mi_Colunas++;
        char cAnterior = ' ';
        if (i != 0)
                cAnterior = Infix.GetAt(i-1);
        if (cAnterior == '~')
                ms_Saida += cAnterior;
        bool    bAchou= false;
        POSITION nPos= mc_ListaVar.GetHeadPosition();
        if (nPos != NULL)
        {
                char    cAux;
                while (nPos != NULL)
                {
                        cAux = mc_ListaVar.GetAt (nPos);
                        if (cAux == Tipo)
                        {
                                bAchou = true;
                                break;
                        }
                        mc_ListaVar.GetNext (nPos);
                }
        }
        if (!bAchou)
        {
                ms_Proposicao += Tipo;
                mc_ListaVar.AddTail (Tipo);
        }
        break;
```

```
        }
        if (bNegacao)
                ms_Saida += '~';
        while (iNegacao != 0)
        {
                ms_Saida += '~';
                iNegacao--;
        }
        if(simbolos.size() != 0)
        {
                Tipo = simbolos.top();
                ms_Saida += Tipo;
                simbolos.pop();
        }
        simbolos.empty();
}

void CSemantico::ValidaFormula()
{
        CString sValidade;
        pArray = NULL;
        mc_Lista.RemoveAll();
        mi_Linhas = 2;
        int iVar = mc_ListaVar.GetCount();
        int iLin = 0;
        int iCol = 0;
        int iAux = 1;
        char cTipo = ' ';
        bool bPassou = true;

        //Obtem o número de linhas que a tabela terá para a combinação
        //dos valores lógicos (T e F)
        for(int i = 1; i < iVar;)
        {
                mi_Linhas = 2 * mi_Linhas;
                i++;
        }
        // Soma-se mais uma linha para as proposições
        mi_Linhas++;

        //Cria-se uma matriz para armazenar os valores e em seguida
      //zera a tabela.
        pArray = new char * [mi_Linhas];
        memset( pArray, 0, sizeof(char *)*mi_Linhas);
        for( iLin = 0; iLin < mi_Linhas; iLin++ )
        {
                pArray[iLin] = new char [mi_Colunas];
                for( iCol = 0; iCol < mi_Colunas; iCol++ )
                        pArray[iLin][iCol] = 0;
        }

        //Coloca-se as proposições na primeira linha
        for( iLin = 0; iLin == 0; iLin++ )
        {
                for( iCol = 0; iCol < mi_Colunas; iCol++ )
                {
                        cTipo = ms_Saida.GetAt(iCol);
                        pArray[iLin][iCol] = cTipo;
                }
        }
        iAux = (mi_Linhas - 1)/2;
        iCol = 0;
        iLin = 0;
        std::stack<char> Resultado;
        Resultado.empty();
        cTipo = ' ';
        char cValor = ' ';
        bool bAchou = false;
        int iContaCol;

        //Em seguida atribui-se os valores lógicos para cada proposição
        int iTamanho = ms_Saida.GetLength();

        for( int x = 0;  x < iTamanho; x++ )
```

```
                cTipo = ms_Saida.GetAt(iCol);

                //Se for um conectivo é atribuido 'N' para que mais tarde
        {     //seja obtido seu valor
                if ( (cTipo == '|') || (cTipo == '&') || (cTipo == '-')
                        || (cTipo == '<') ||(cTipo == '~') )
                {
                        for( iLin = 1; iLin < mi_Linhas; iLin++ )
                                pArray[iLin][iCol] = 'N';
                        iCol++;
                        continue;
                }
                else
                {
                        bAchou  = false;
                        iContaCol = 0;
                        //Verifica se há proposições iguais e atribui os
                    //mesmo valores lógicos
                        POSITION nPos  = mc_Lista.GetHeadPosition ();
                        if (nPos != NULL)
                        {
                                char    cAux = ' ';
                                while (nPos != NULL)
                                {
                                        cAux = mc_Lista.GetAt (nPos);
                                        if (cAux == cTipo)
                                        {
                                                int iTamanho = ms_Saida.GetLength();
                                                //Verifica qual a posição em q a
                                  //proposição é igual,
                                                //assim obtem-se a coluna que se
                                  //deve copiar os valores
                                                for( int x = 0;  x < iTamanho; x++ )
                                                {
                                                        if (cAux == ms_Saida.GetAt(x))
                                                        {
                                                                iContaCol = x;
                                                                break;
                                                        }
                                                }

                                                for( iLin = 1; iLin < mi_Linhas; iLin++ )
                                                        pArray[iLin][iCol] =
pArray[iLin][iContaCol];
                                                bAchou = true;
                                        }
                                        mc_Lista.GetNext (nPos);
                                }
                        }
                        if (!bAchou)
                        {
                                mc_Lista.AddTail (cTipo);
                                //Neste ponto é atribuido as combinações
                              //lógicas da diferentes proposições
                                for( iLin = 1; iLin < mi_Linhas; iLin++ )
                                {
                                        if (iAux == 1)
                                        {
                                                if (bPassou)
                                                {
                                                        pArray[iLin][iCol] = 'T';
                                                        bPassou = false;
                                                }
                                                else
                                                {
                                                        pArray[iLin][iCol] = 'F';
                                                        bPassou = true;
                                                }
                                        }
                                        else
                                                if (bPassou)
                                                {
                                                        pArray[iLin][iCol] = 'T';
                                                        for(int x = 1; x < iAux; x++)
```

```
                                                iLin++;
                                                pArray[iLin][iCol] = 'T';
                                        }
                                        bPassou = false;
                                }
                                else
                                {
                                        pArray[iLin][iCol] = 'F';
                                        for(int x = 1; x < iAux; x++)
                                        {
                                                iLin++;
                                                pArray[iLin][iCol] = 'F';
                                        }
                                        bPassou = true;
                                }
                        }
                        iAux = iAux/2;
                }
        }
        iCol++;
}

//Através da notação posfix é obtido os resultados da fórmula
for( iLin = 1; iLin < mi_Linhas; iLin++ )
{
        int y = 0;
        iVar = mc_ListaVar.GetCount();
        for( iCol = 0; iCol < mi_Colunas; iCol++ )
        {
                char cValorX = ' ';
                char cValorY = ' ';
                cTipo = ms_Saida.GetAt(y);

                //Quando for um conectivo desempilha os dois ultimos
         //valores para X e Y
                //Obtem-se o valor desta operação através da função
         //ValorConectivo() e empilha o resultado
                //Caso for uma proposição empilha o valor desta
                if ( (cTipo == '|') || (cTipo == '&')
               || (cTipo == '-') || (cTipo == '<') )
                {
                        if(Resultado.size() != 0)
                        {
                                cValorY = Resultado.top();
                                Resultado.pop( );
                        }
                        if(Resultado.size() != 0)
                        {
                                cValorX = Resultado.top();
                                Resultado.pop( );
                        }
                        cValor=ValorConectivo(cTipo, cValorX, cValorY);
                        Resultado.push(cValor);
                        pArray[iLin][iCol] = cValor;
                }
                else
                        if (cTipo == '~')
                        {
                                if(Resultado.size() != 0)
                                {
                                        cValorX = Resultado.top();
                                        Resultado.pop( );
                                }
                                if (cValorX == 'T')
                                        cValor = 'F';
                                else
                                        if (cValorX == 'F')
                                                cValor = 'T';
                                Resultado.push(cValor);
                                pArray[iLin][iCol] = cValor;
                        }
                        else
                        {
                                cValor = pArray[iLin][iCol];
```

```
                                }
                                y++;
                        }
                                Resultado.push(cValor);
                }
                int nTrue = 0,
                        nFalse = 0;

                //Verifica-se a ultima coluna onde esta definirá
                //se é uma tautologia, contraditória ou satisfatível.
                iCol = mi_Colunas - 1;
                mi_True = 0;
                mi_False = 0;
                for( iLin = 0; iLin < mi_Linhas; iLin++ )
                {
                        if (pArray[iLin][iCol] == 'T')
                                mi_True++;
                        if (pArray[iLin][iCol] == 'F')
                                mi_False++;
                }
                Resultado.empty();
        }

        char CSemantico::ValorConectivo(char Tipo, char ValorX, char ValorY)
        {
                char cVlConectivo = ' ';
                switch (Tipo)
                {
                case '|':
                        {
                                if ((ValorX == 'T') & (ValorY == 'T'))
                                        cVlConectivo = 'T';
                                else
                                        if ((ValorX == 'T') & (ValorY == 'F'))
                                                cVlConectivo = 'T';
                                        else
                                                if ((ValorX == 'F') & (ValorY == 'T'))
                                                        cVlConectivo = 'T';
                                                else
                                                        if ((ValorX == 'F') & (ValorY == 'F'))
                                                                cVlConectivo = 'F';
                        }
                        break;
                case'&':
                        {
                                if ((ValorX == 'T') & (ValorY == 'T'))
                                        cVlConectivo = 'T';
                                else
                                        if ((ValorX == 'T') & (ValorY == 'F'))
                                                cVlConectivo = 'F';
                                        else
                                                if ((ValorX == 'F') & (ValorY == 'T'))
                                                        cVlConectivo = 'F';
                                                else
                                                        if ((ValorX == 'F') & (ValorY == 'F'))
                                                                cVlConectivo = 'F';
                        }
                        break;
                case'-':
                        {
                                if ((ValorX == 'T') & (ValorY == 'T'))
                                        cVlConectivo = 'T';
                                else
                                        if ((ValorX == 'T') & (ValorY == 'F'))
                                                cVlConectivo = 'F';
                                        else
                                                if ((ValorX == 'F') & (ValorY == 'T'))
                                                        cVlConectivo = 'T';
                                                else
                                                        if ((ValorX == 'F') & (ValorY == 'F'))
                                                                cVlConectivo = 'T';
                        }
                        break;
                case'<':
                        {
```

```
                        cVlConectivo = 'T';
                else
                if ((ValorX == 'T') & (ValorY == 'F'))
                        cVlConectivo = 'F';
                else
                        if ((ValorX == 'F') & (ValorY == 'T'))
                                cVlConectivo = 'F';
                        else
                                if ((ValorX=='F') & (ValorY=='F'))
                                        cVlConectivo = 'T';
                }
                break;
        }
        return cVlConectivo;
}
```