



COORDENAÇÃO DE ENGENHARIA DA COMPUTAÇÃO

RODRIGO VIEIRA DA SILVA

**FRAMEWORK PARA CONSTRUÇÃO DE COMPILADORES COM
CONCEITOS FUZZY**

Sorocaba/SP

2015

Rodrigo Vieira da Silva

**FRAMEWORK PARA CONSTRUÇÃO DE COMPILADORES COM
CONCEITOS FUZZY**

Trabalho de Conclusão de Curso apresentado à
Faculdade de Engenharia de Sorocaba – FACENS,
como parte dos pré-requisitos para obtenção do
título de Engenheiro da Computação.

Orientador: Marcos Maurício Lombardi Pellini Fernandes

Sorocaba/SP

2015

FICHA CATALOGRÁFICA
ELABORADA PELA “BIBLIOTECA FACENS”

S586f

Silva, Rodrigo Vieira da.

Framework para construção de compiladores com conceitos fuzzy / por
Rodrigo Vieira da Silva – Sorocaba, SP: {s.n.}, 2015.
65f.; 29 cm.

Trabalho de Conclusão de Curso (Graduação) – Faculdade de Engenharia
de Sorocaba, Coordenadoria de Engenharia da Computação - Curso de
Engenharia da Computação, 2015.

Orientador: Marcos Maurício Lombardi Pellini Fernandes.

1. Compiladores (Programas de computador). 2. Inteligência Artificial.
I. Autor. II. Faculdade de Engenharia de Sorocaba. III. Título.

CDU 004.4'422

FRAMEWORK PARA CONSTRUÇÃO DE COMPILADORES COM CONCEITOS FUZZY

Trabalho de Conclusão de Curso apresentado à
Faculdade de Engenharia de Sorocaba, como
exigência parcial para obtenção do Diploma de
Graduação em Engenharia da Computação.

Comissão examinadora:

Prof. Marcos Maurício Lombardi Pellini
Fernandes

Ma. Renata Corrêa Pimentel

Lincoln Kovalski

Coordenadora:

Dr^a Andréa Lúcia Braga Vieira Rodrigues

Sorocaba/SP

2015

Dedico este trabalho a minha esposa Jaqueline, família, amigos e professores que me guiaram por este longo e árduo caminho.

AGRADECIMENTOS

De proêmio, agradeço à Providência Divina, pelas brilhantes oportunidades que colocou em minha vida.

Outrossim, agradeço reverentemente a todos aqueles que foram e aos que ainda são meus professores, em especial ao Marcos Maurício, cujas orientações deram forma a este trabalho, e a Renata Pimentel, pela introdução ao mundo dos compiladores.

Aquele que adquire entendimento ama sua alma.
Aquele que conserva a inteligência acha o bem.

Provérbios 19:8

O que o amor ousa, ao amor é permitido.

William Shakespeare, Romeu e Julieta, Segundo
Ato, Cena II

À medida que a complexidade aumenta, as
declarações precisas perdem relevância e as
declarações relevantes perdem precisão.

Lotfi A. Zadeh

RESUMO

Silva, R. V. Framework para construção de compiladores com conceitos fuzzy. Sorocaba, 2015, 65p. Trabalho de Conclusão de Curso (Graduação) – Curso de Engenharia da Computação, Faculdade de Engenharia de Sorocaba. Sorocaba, 2015.

Embora existam diversas ferramentas que auxiliam na definição de compiladores, a aplicação de lógica nebulosa em compiladores ainda continua pouco explorada. Este trabalho consiste em demonstrar uma ferramenta que auxilia na definição e elaboração de um compilador que, além de suas características básicas, conta com aplicações de lógica *fuzzy* em diferentes partes do processo de compilação e para o correto entendimento, é abordado os principais conceitos de lógica nebulosa que podem ser aplicados à compiladores.

Palavras chave: *Compiladores (Programas de computador). Inteligência Artificial.*

ABSTRACT

Silva, R. V. Framework para construção de compiladores com conceitos fuzzy. Sorocaba, 2015, 57p. Trabalho de Conclusão de Curso (Graduação) – Curso de Engenharia da Computação, Faculdade de Engenharia de Sorocaba. Sorocaba, 2015.

While there are several tools that assist in setting compilers, applications of fuzzy logic in compilers still remains little explored. This work is to demonstrate a tool that assists in the definition and development of a compiler that in addition to its basic characteristics, features fuzzy logic applications in different parts of the build process and the right view, it is approached the main concepts of logic fuzzy that can be applied to compilers.

Keywords: *Compilers (Computer programs). Artificial intelligence.*

LISTA DE ABREVIATURAS E SIGLAS

AFD – Autômato Finito Determinístico

AFN – Autômato Finito Nebuloso

FREGEX – Expressão Regular *Fuzzy*

GLC – Gramática Livre de Contexto

GLCP – Gramática Livre de Contexto Probabilística

REGEX – Expressão Regular

XML – Extensible Markup Language

LISTA DE FIGURAS

Figura 2.1 – Etapas de um compilador.....	18
Figura 2.2. – Autômato Finito Determinístico	20
Figura 2.3 – Grafo de transição AFD.....	20
Figura 2.4 – Exemplo de expressão regular.....	21
Figura 2.5 – Definição Exemplo 1	22
Figura 2.6 – Grafo Transição Exemplo 1	22
Figura 2.7 – Exemplo gramática.....	24
Figura 2.8 – Algoritmos de Análise Sintática	25
Figura 3.1 – Representação universo U discreto	27
Figura 3.2 – Representação conjunto A	28
Figura 3.3 – Representação do conjunto A de forma gráfica.	28
Figura 3.4 – Representação pertinência dos números no conjunto P	29
Figura 3.5 – Representação do conjunto P de forma gráfica.	29
Figura 3.6 – Representação dos conjuntos P, M e G.....	30
Figura 3.7 – Representação do conjunto Não Grande (NG)	31
Figura 3.8 – Representação do conjunto Grande ou Médio ($G \cup M$)	32
Figura 3.9 – Representação do conjunto Médio E Pequeno ($M \cap P$)	33
Figura 3.10 – Principais ferramentas e suas características	34
Figura 4.1 – AFN para palavra bau	36
Figura 4.2 – Processamento de um AFN da palavra “aa”	36
Figura 4.3 – Exemplo de uma gramática <i>fuzzy</i>	38
Figura 4.4 – Algoritmo LR(1) modificado	38
Figura 4.5 – Exemplo de uma gramática probabilística.....	39
Figura 4.6 – Algoritmo LR(1) modificado	40
Figura 5.1 – Diagrama de classe módulo de Autômato <i>Fuzzy</i> e REGEX <i>Fuzzy</i>	42
Figura 5.2 – Aplicação de testes do módulo de autômato <i>fuzzy</i>	43
Figura 5.3 – Pertinência parcial aplicação de testes do módulo de autômato <i>fuzzy</i>	44
Figura 5.4 – Exemplo utilização Classe RecognitionFuzzy	44
Figura 5.5 – Exemplo utilização com definição de norma e conorma	45
Figura 5.6 – Diagrama de classe módulo Gramática Fuzzy.....	45
Figura 5.7 – Diagrama de classe módulo Análise Léxica <i>Fuzzy</i>	46
Figura 5.8 – Análise Léxica sem a técnica de Tokenizer	47
Figura 5.9 – Análise Léxica utilizando a técnica de <i>Tokenizer</i>	48
Figura 5.10 – Análise Léxica utilizando a técnica de Tokenizer	49
Figura 5.11 – Diagrama de classe módulo compilador	50
Figura 5.12 – XML de Configuração do primeiro Exemplo.....	51
Figura 5.13 – Primeiro Exemplo utilização do <i>Framework</i>	52
Figura 5.14 – Tela inicial processamento	53
Figura 5.15 – Árvore sintática gerada (corrigida) a partir do processamento...	54

Figura 5.16 – Autômato intermediário da análise sintática LR(1) <i>fuzzy</i>	55
Figura 5.17 – Tabela de transições e operações algoritmo LR(1).....	56
Figura 5.18 – Tela de demonstração do processamento léxico	57
Figura 5.19 – Demonstração de processamento com sentença “get”	58
Figura 5.20 – Demonstração de processamento com sentença “go get the ball”	59
Figura 5.21 – Demonstração de processamento com sentença “ogo geet balll”	59
Figura 5.22 – XML de configuração exemplo 2	60
Figura 5.23 – Processamento cadeia “geeeet ball dog”	61
Figura 6.1 – Tela principal da aplicação de Consulta.....	62

SUMÁRIO

1	INTRODUÇÃO.....	16
2	PRINCÍPIOS DE COMPILADORES	17
2.1	Conceitos	17
2.1.1	Análise Léxica.....	18
2.1.1.1	Linguagens Regulares	19
2.1.1.2	Autômatos Finitos Determinísticos.....	19
2.1.1.3	Expressões Regulares.....	21
2.1.2	Análise Sintática	23
2.1.2.1	Gramática Livres de Contexto	23
2.1.2.2	Técnicas	24
2.1.3	Outras Análises.....	25
2.2	Ferramentas	26
3	LÓGICA NEBULOSA (FUZZY)	27
3.1	Conceitos	27
3.1.1	Operações em conjuntos Nebulosos	30
3.1.1.1	Complemento.....	30
3.1.1.2	União.....	31
3.1.1.3	Intersecção	32
3.2	Ferramentas e Aplicações	33
4	LÓGICA NEBULOSA APLICADO A COMPILADORES	35
4.1	Análise Léxica com conceitos <i>fuzzy</i>.....	35
4.1.1	Autômato Finito <i>Fuzzy</i>	35
4.2	Análise Sintática com conceitos <i>fuzzy</i>	37
4.2.1	Gramática <i>Fuzzy</i> Livre de Contexto.....	37
4.2.2	Gramática Livre de Contexto Probabilística	39
5	FRAMEWORK PARA CONSTRUÇÃO DE COMPILADORES COM CONCEITOS FUZZY	41
5.1	Autômato e REGEX <i>Fuzzy</i>.....	41
5.2	Gramática <i>Fuzzy</i>	45
5.3	Análise Léxica <i>Fuzzy</i>.....	46
5.4	Análise Sintática <i>Fuzzy</i>	49
5.5	Compilador <i>Fuzzy</i>.....	49
5.6	Exemplo de utilização da ferramenta.....	50
6	FERRAMENTA DE CONSULTA EM BANCO DE DADOS FACILITADA	62

7	CONCLUSÃO	63
7.1	Perspectivas Futuras.....	63
	REFERÊNCIAS.....	64

1 INTRODUÇÃO

Um compilador é essencial para maior eficiência na construção de um programa de computador, para o seu desenvolvimento é necessário o entendimento de diversas etapas bem definidas que, podem ser generalizadas e disponibilizadas como ferramenta para facilitar a elaboração de um *software* capaz de transformar um código de origem em um código de destino.

A lógica nebulosa ou lógica *fuzzy* é uma extensão da lógica clássica, admitindo valores intermediários entre 0 e 1 se torna capaz de ultrapassar as barreiras delimitadas pelos conceitos discretos de pertence ou não pertence.

Este trabalho possui como principal objetivo a união entre os dois conceitos, aplicando a lógica *fuzzy* nas etapas de análise léxica e análise sintática de um compilador.

Para o correto entendimento o primeiro capítulo realiza um estudo sobre os compiladores convencionais e suas etapas, já o segundo capítulo aborda a lógica nebulosa e seus principais conceitos, o terceiro capítulo aborda como a lógica *fuzzy* pode ser aplicada nas diferentes etapas de um compilador e em seguida é apresentada a ferramenta desenvolvida e a sua correta utilização.

Na continuidade é abordado o desenvolvimento de uma aplicação de consulta de dados que utiliza todas as etapas da ferramenta implementada e, por fim, apresentado uma conclusão sobre as diferentes etapas desenvolvidas.

2 PRINCÍPIOS DE COMPILADORES

De forma abrangente um compilador converte um código fonte em um código destino. Os princípios e técnicas de construção de compiladores são utilizadas em diversas áreas de aplicação do conhecimento de um profissional envolvido com as disciplinas de computação. Com o aprendizado de técnicas básicas é possível abordar uma grande variedade de problemas quanto a tradutores de linguagens e máquinas. (AHO; LAN; ULLMAN, 1995)

Este capítulo abordará algumas dessas técnicas e conceitos que foram utilizados para o desenvolvimento desse projeto.

2.1 Conceitos

Um compilador é essencial para maior eficiência na construção de um programa, por possuir etapas bem definidas é possível a sua modularização e generalização em diversos itens como demonstra a figura 2.1. (AHO; LAN; ULLMAN, 1995)

Figura 2.1 – Etapas de um compilador



Fonte: AHO, LAN E ULLMAN (1995)

2.1.1 Análise Léxica

Análise Léxica, análise linear ou ainda *scanning* (esquadrinhamento) em um compilador é responsável por ler o código fonte e converter em um fluxo de *tokens* (palavras e tipos que compõe o texto) que será propagado para a próxima etapa, ou seja, recebe uma sequência de caracteres e gera uma lista sequencial de palavras chaves, pontuação e nomes ignorando comentários de código e espaços em brancos. (AHO; LAN; ULLMAN, 1995), (RIGO, 2015).

Esse processamento é semelhante ao reconhecimento de cada palavra e sua classe gramatical – verbos, adjetivos e substantivos – de uma linguagem natural. Para

realizar esse procedimento normalmente são utilizados expressões regulares e autômatos finitos. (RICARTE, 2008).

2.1.1.1 Linguagens Regulares

Para o correto entendimento de uma linguagem é necessário compreender alguns conceitos.

O primeiro conceito importante é o alfabeto, que se trata de um conjunto finito de símbolos, em segundo, uma cadeia que é uma sequência de símbolos que pertencem a um alfabeto, por exemplo “aoia” é uma cadeia válida sobre o alfabeto das vogais (a, e, i, o, u). Finalmente uma linguagem corresponde a um conjunto de cadeias sobre o alfabeto. Neste trabalho será apenas abordado a linguagem regulares que tem como característica possuir um autômato finito equivalente. (MACIEL, 2006).

Para uma linguagem ser regular deve possuir as seguintes propriedades (MACIEL, 2006):

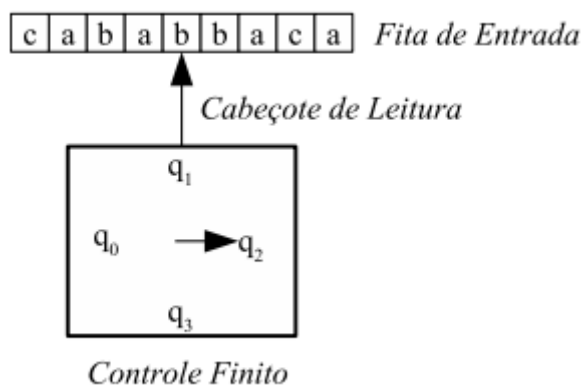
- Se a linguagem é igual a um conjunto vazio ou a linguagem é igual ao conjunto de qualquer símbolo pertencente ao alfabeto.
- L1 e L2 são linguagens regulares se uma linguagem regular é obtida a partir da união entre as duas linguagens.
- L1 e L2 são linguagens regulares se uma linguagem regular é obtida pela concatenação entre as duas linguagens.
- Se L1 é regular então a linguagem obtida a partir da concatenação de zero ou mais cadeias da linguagem L1 também é regular.

2.1.1.2 Autômatos Finitos Determinísticos

O autômato finito determinístico (AFD) é o modelo computacional existente menos complexo, são capazes de processar informações, recebendo uma entrada e exibindo uma saída, não possuem uma memória auxiliar e apesar disso ele é totalmente adequado à função de reconhecimento de cadeias de texto. (MACIEL, 2006).

A Figura 2.2 demonstra a composição de um AFD, fita de entrada, cabeçote de leitura e um controle finito que é um conjunto de estados distintos.

Figura 2.2. – Autômato Finito Determinístico



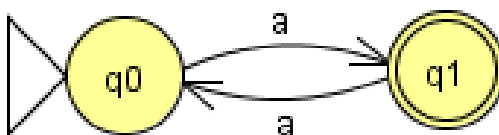
Fonte: MACIEL, 2006.

Um AFD é formado por uma quintupla com a seguinte definição (AHO; LAN; ULLMAN, 1995):

- 1) Conjunto finito de estados.
- 2) Conjunto de símbolos de entrada.
- 3) Função de transição, que consiste na escolha de um próximo estado a partir de um símbolo e de um estado atual.
- 4) Estado inicial
- 5) Conjunto de estados finais

Pode-se representar um AFD através de um grafo de transição, como demonstra a figura 2.3.

Figura 2.3 – Grafo de transição AFD



2.1.1.3 Expressões Regulares

Entendido os conceitos de linguagens regulares e de um AFD é possível aplicá-los para definir uma expressão regular (REGEX) e realizar a sua validação computacionalmente.

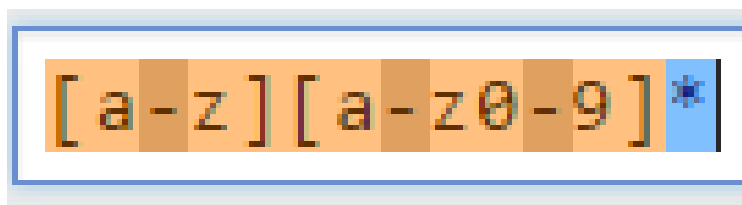
Uma expressão regular representa um padrão de cadeia de caracteres, ela é definida pelo conjunto de cadeias que “valida”. (LOUDEN, 2004).

Existem três principais operações em expressões regulares (LOUDEN, 2004):

1. Operação “OU”, normalmente representada pelo caractere “|” (barra vertical) ou em formato de lista entre “[]” (colchetes);
2. Concatenação ou sequência;
3. Repetição, que normalmente é representada pelo caractere “*”;;

A seguir um exemplo onde são demonstrados uma expressão regular, sua linguagem regular correspondente e o grafo do autômato determinístico gerado a partir da expressão.

Figura 2.4 – Exemplo de expressão regular



Na figura 2.4 é possível observar uma expressão regular que valida uma cadeia que comece com um caractere que esteja entre o intervalo de “a-z” e que termine com zero ou mais caracteres entre “a-z” ou “0-9”.

Figura 2.5 – Definição Exemplo 1

AFD = (S, Q, d, q0, F)
S = {a, b, c, d, ..., z, 0, 1, 2, 3, ..., 9}
Q = {q0, q1}
D =

	a-z	0-9
q0	q1	-
q1	q1	q1

F = {q1}

A figura 2.5 monta a quintupla que define um autômato finito determinístico onde:

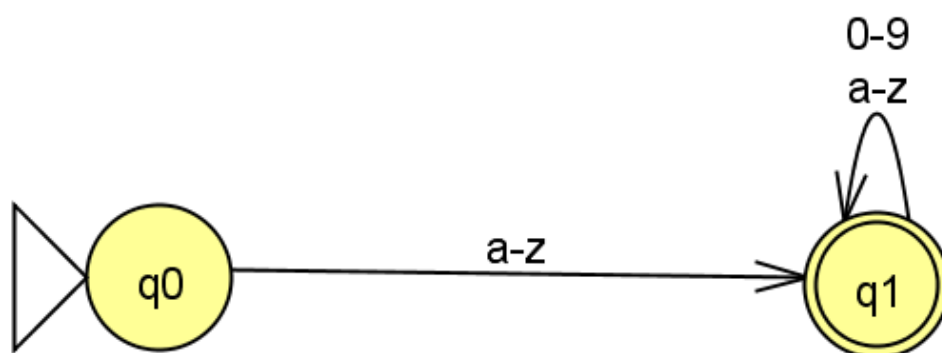
“S” é o alfabeto de símbolos que contempla nos intervalos de ‘a’ a ‘z’ e de ‘0’ a ‘9’.

“Q” é o conjunto de estados que formam o autômato.

“D” é a função de transição, que pode ser definida em formato de tabela, onde as linhas representam os estados de origem que ao consumir um símbolo da coluna da linha superior passa para um novo estado da célula correspondente.

Finalmente “F” é o conjunto de estados finais representado pelo estado q1.

Figura 2.6 – Grafo Transição Exemplo 1



Na figura 2.6 é possível observar o grafo de transição do AFD correspondente a expressão regular do exemplo 1.

No exemplo 1 podemos validar um tipo de *token* de identificador (nomes de variáveis ou métodos), muito utilizado na maioria das linguagens de programação. (LOUDEN, 2004).

Existem diversas técnicas e algoritmos para minimização e simplificação de autômatos finitos determinísticos que não serão apresentados nesse trabalho, já no capítulo 5 será descrito as diversas alterações realizadas nos algoritmos de processamento e conversão de uma linguagem regular para um autômato.

2.1.2 Análise Sintática

O analisador sintático é o responsável por gerar a árvore sintática a partir do fluxo de *tokens* que o analisador léxico montou, emitindo qualquer erro sintático encontrado durante o processamento. Para realizar o seu papel o analisador necessita de uma gramática representativa. (AHO; LAN; ULLMAN, 1995).

Existem diversas classificações de gramáticas segundo a classificação de Chomsky, neste trabalho será abordada as gramáticas livres de contexto. (AHO; LAN; ULLMAN, 1995).

2.1.2.1 Gramática Livres de Contexto

Uma gramática livre de contexto é formada por uma quádrupla com a seguinte definição (AHO; LAN; ULLMAN, 1995):

1. Conjuntos de variáveis ou não-terminais
2. Conjunto de símbolos terminais
3. Conjunto de produções, onde uma produção contém um lado esquerdo, composto por uma variável e um lado direito, que pode conter um conjunto de variáveis e terminais.
4. Uma variável inicial

Figura 2.7 – Exemplo gramática

$$\begin{aligned}
 G_2 &= (\{S, A, B, C\}, \{0, 1\}, P_2, S): \\
 P_2 &= \{S \rightarrow 0A0 \mid 1B1 \mid BB \\
 &\quad A \rightarrow C \\
 &\quad B \rightarrow S \mid A \\
 &\quad C \rightarrow S \mid \epsilon\}
 \end{aligned}$$

Fonte: SAKATA, 2015.

A figura 2.7 demonstra a formalização de uma gramática, onde o conjunto de variáveis é composto por 'S', 'A', 'B', 'C', o conjunto de símbolos terminais possui os números 0 e 1, o conjunto P2 representa as diferentes regras de derivações e finalmente o S é a variável inicial.

As GLCs (Gramáticas Livre de Contexto) são capazes de representar a sintaxe de uma linguagem de programação e, assim, nessa etapa é possível a validação e geração da árvore de derivação. (AHO; LAN; ULLMAN, 1995).

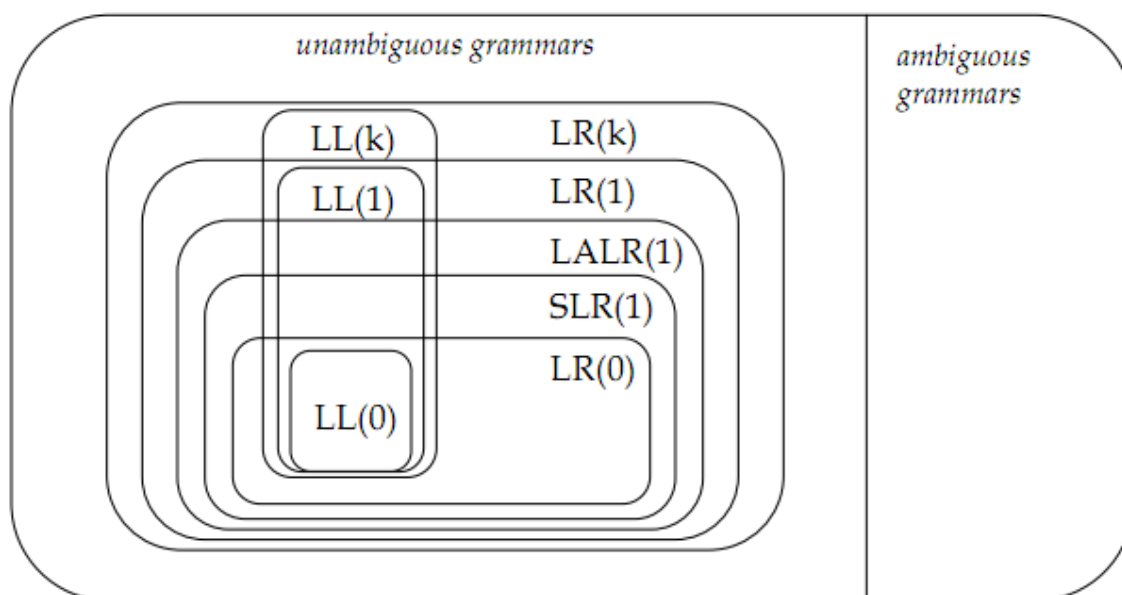
2.1.2.2 Técnicas

A análise sintática pode ser realizada de forma ascendente ou descendente, a figura 2.8 categoriza os diversos algoritmos para validação sintática através de uma gramática.

Figura 2.8 – Algoritmos de Análise Sintática

LL(1) versus LR(k)

A picture is worth a thousand words:



Fonte: APPEL, GINSBURG, 1998

Ainda na figura 2.8 é possível visualizar a abrangência dos algoritmos quanto as gramáticas suportadas, o algoritmo mais utilizado nos compiladores atuais é a LALR(1) que é uma otimização do algoritmo LR(1). (APPEL; GINSBURG, 1998).

Neste trabalho será abordado a LR(1) devido a sua maior abrangência (observado na figura 2.8) e mais fácil adaptação. (APPEL; GINSBURG, 1998).

2.1.3 Outras Análises

Conforme demonstrado na figura 2.1, compiladores possuem diversas etapas bem definidas além dos módulos tratados nesse capítulo, porém sua generalização acaba se tornando muito mais complexa, assim as ferramentas que auxiliam no processo de compilação acabam não abordando essas etapas. (GESSER, 2003).

2.2 Ferramentas

Existem diversas ferramentas que auxiliam nas etapas bem definidas de um compilador, segue alguns exemplos (GESSER, 2003):

LEX: Ferramenta de geração de analisadores léxicos, sendo um dos mais tradicionais geradores. Consiste em um programa que a partir de uma especificação léxica de expressões regulares gera uma rotina de análise léxica em diversas linguagens. (GESSER, 2003).

Microlex: Projeto acadêmico que também possui como entrada expressões regulares e a partir é gerado um analisador léxico em Object Pascal, C++ ou JAVA. (GESSER, 2003).

Yacc (Yet Another Compiler-Compiler): Programa criador de compiladores que consiste em um gerador de analisador sintática, trabalha em conjunto com a ferramenta LEX. (GESSER, 2003).

3 LÓGICA NEBULOSA (FUZZY)

Jan Lukasiewicz (1787-1956), lógico polonês, em 1920 introduziu as primeiras noções de lógica com conceitos vagos, adotando conjuntos com graus de pertinência 0, 0.5 e 1 e, mais tarde, números infinitos no intervalo de 0 a 1. (ABAR, 2015).

A palavra “*fuzzy*”, em inglês, pode ter vários significados de acordo com o contexto, porém o conceito de incerteza e vago sempre está presente. O termo mais aceito na engenharia é nebuloso ou difuso. (REZENDE, 2006).

A primeira publicação sobre lógica *fuzzy* foi proposta por Zadeh em 1965, a lógica nebulosa veio como alternativa para representação de informações vagas ou imprecisas e pode ser classificada como uma área da Inteligência Artificial (IA). A teoria dos conjuntos nebulosos é considerada como extensão da teoria dos conjuntos e como as informações são processadas pode ser vista como uma extensão da lógica clássica. (LOPES; PINHEIRO; SANTOS, 2014).

3.1 Conceitos

A teoria clássica de conjuntos trata as classes de objetos e as suas relações em um universo limitado, bem definido. A figura 3.1 define um universo U discreto que possui todos os números entre -10 e 10 do conjunto Z dos números inteiros. (REZENDE, 2006).

Figura 3.1 – Representação universo U discreto

$$U: \{x \in Z \mid \text{módulo}(x) \leq 10\}$$

Fonte: (REZENDE, 2006)

Os elementos de um mesmo universo podem ser agrupados por suas características semelhantes, na figura 3.2 representamos um conjunto A obtido a partir do universo U de discurso. (REZENDE, 2006)

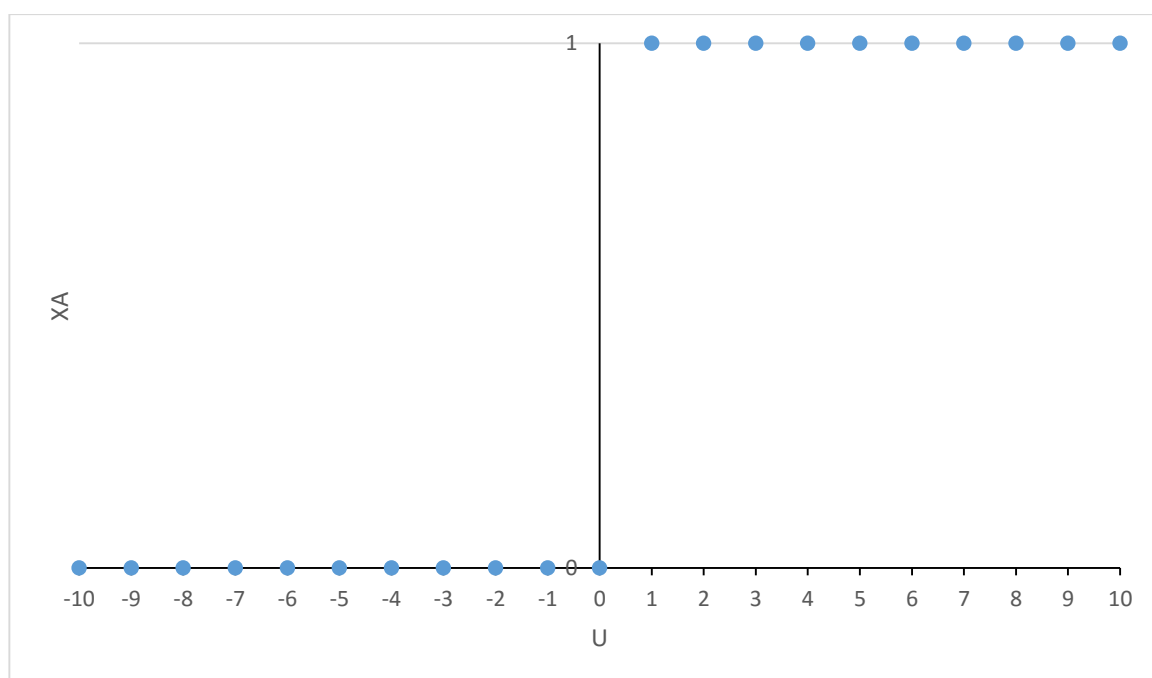
Figura 3.2 – Representação conjunto A

$$A: \{x \in U \mid x > 0\}$$

Fonte: (REZENDE, 2006)

Uma outra forma de representação de um conjunto é através de um gráfico como demonstra a figura 3.3. (CAUSSEY, 1994).

Figura 3.3 – Representação do conjunto A de forma gráfica.



Fonte: (REZENDE, 2006)

A figura 3.3 demonstra também que os números do conjunto A possuem pertinência total (1), porém no mundo real e em grande parte das aplicações de interesse na engenharia existem propriedades que são vagas, incertas ou imprecisas. A lógica nebulosa, como extensão da lógica clássica, admite valores intermediários entre a pertinência mínima (0) e máxima (1). (REZENDE, 2006). (ABAR, 2015).

A partir do mesmo universo de discurso U pode-se obter um conjunto nebuloso denotado por P, na figura 3.4 observa-se o grau de pertinência dos números no conjunto: (REZENDE, 2006)

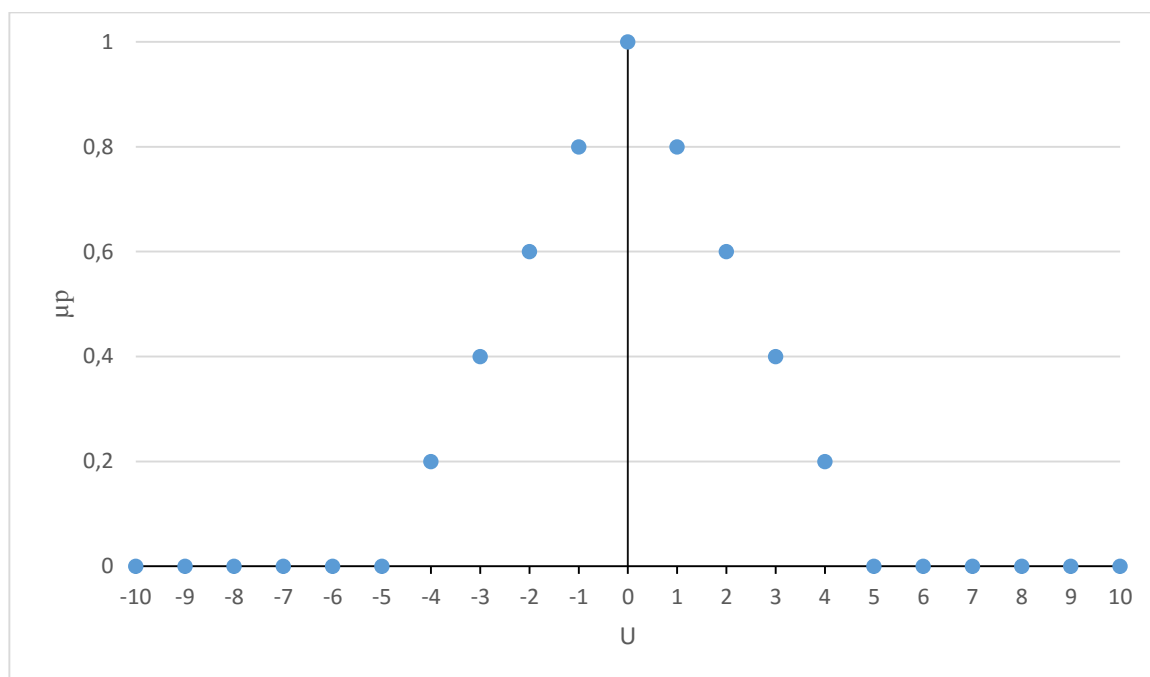
Figura 3.4 – Representação pertinência dos números no conjunto P

$$\mu_P(x) = \begin{cases} 0,0 & , se \text{ módulo}(x) > 5 \\ \frac{5 - \text{módulo}(x)}{5} & , se \text{ módulo}(x) \leq 5 \end{cases}$$

Fonte: (REZENDE, 2006)

Do mesmo modo que os conjuntos clássicos, podemos representar graficamente o conjunto nebuloso P, conforme demonstrado na figura 3.5. (REZENDE, 2006).

Figura 3.5 – Representação do conjunto P de forma gráfica.



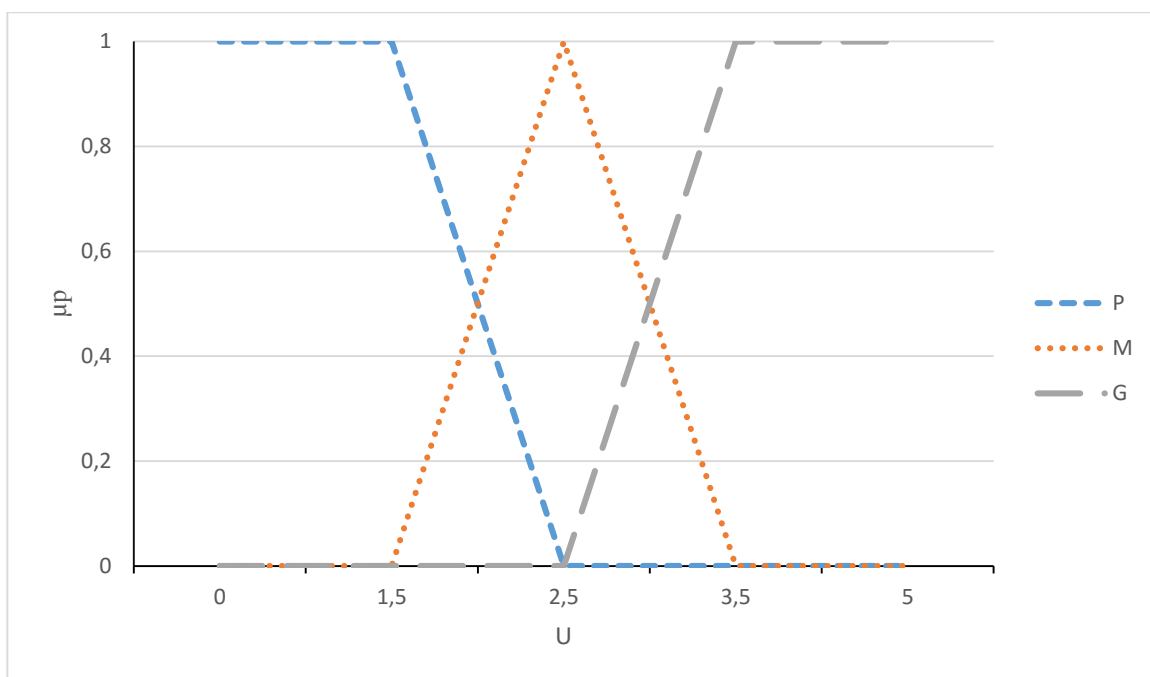
Fonte: (REZENDE, 2006)

Pode-se verificar a diferença entre as fronteiras bem definidas da lógica clássica quanto a um elemento (pertence ou não pertence) e os critérios e graus de pertinência para cada conjunto da Lógica Nebulosa. (ABAR, 2015).

3.1.1 Operações em conjuntos Nebulosos

A partir de três conjuntos nebulosos denominados pequeno (P), médio (M) e grande (G) em um universo de discurso real $U: [0,5]$, demonstrados na figura 3.6, serão apresentadas as operações básicas nebulosas. (REZENDE, 2006).

Figura 3.6 – Representação dos conjuntos P, M e G

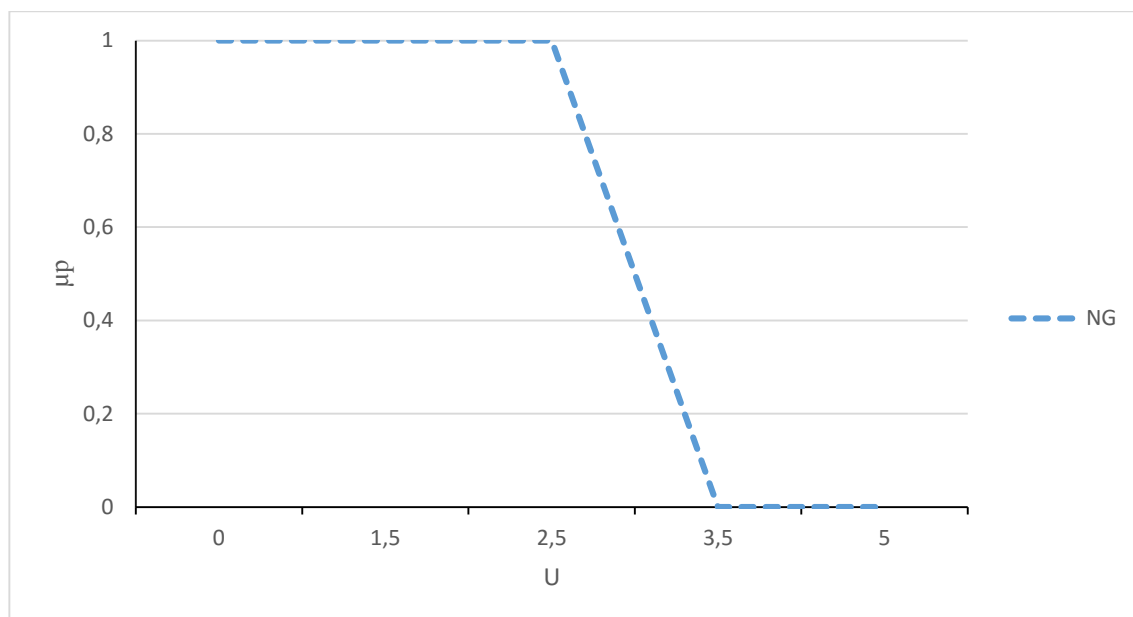


Fonte: (REZENDE, 2006)

3.1.1.1 Complemento

O Complemento de um conjunto nebuloso A pode ser representado por $\neg A$. Na figura 3.7 demonstra o conjunto resultante da operação de complemento do conjunto grande, ou seja, o conjunto não grande sobre o mesmo universo. (REZENDE, 2006). (ABAR, 2015).

Figura 3.7 – Representação do conjunto Não Grande (NG)



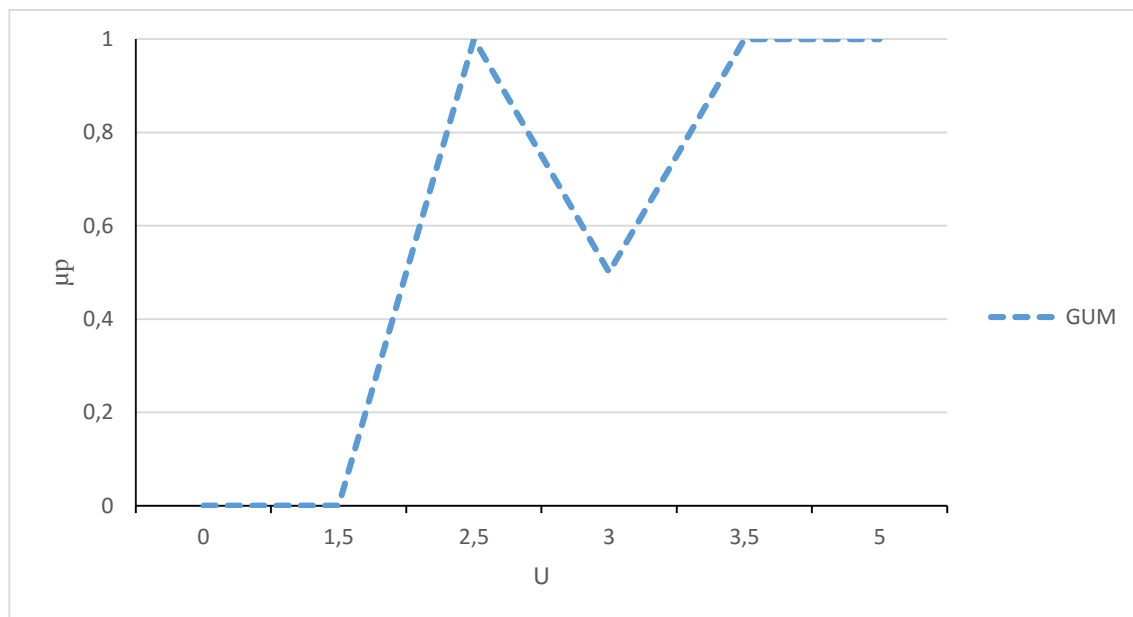
Fonte: (REZENDE, 2006)

3.1.1.2 União

A união entre dois conjuntos C e D pode ser definida por $C \cup D$, Zadeh propôs a seguinte conorma para uma operação de união S. (REZENDE, 2006).

1. Comutatividade: $S(C, D) = A(C, D)$;
2. Associatividade: $S(C, S(D, E)) = S(S(C, D), E)$
3. Monotonicidade: se $C \leq D$ e $E \leq F$, então $S(C, E) \leq S(D, F)$
4. Coerência nos contornos: $S(C, 1) = 1$ e $S(C, 0) = C$

Dessa forma a união corresponde sempre ao conectivo “OU”, representado na figura 3.8 com o conjunto “grande ou médio”, utilizando como conorma a função “máximo”.

Figura 3.8 – Representação do conjunto Grande ou Médio ($G \cup M$)

Fonte: (REZENDE, 2006)

3.1.1.3 Intersecção

A intersecção entre dois conjuntos C e D pode ser definida por $C \cap D$, Zadeh propôs a seguinte norma para uma operação de intersecção S. (REZENDE, 2006). (ABAR, 2015).

Comutatividade: $S(C, D) = A(C, D)$;

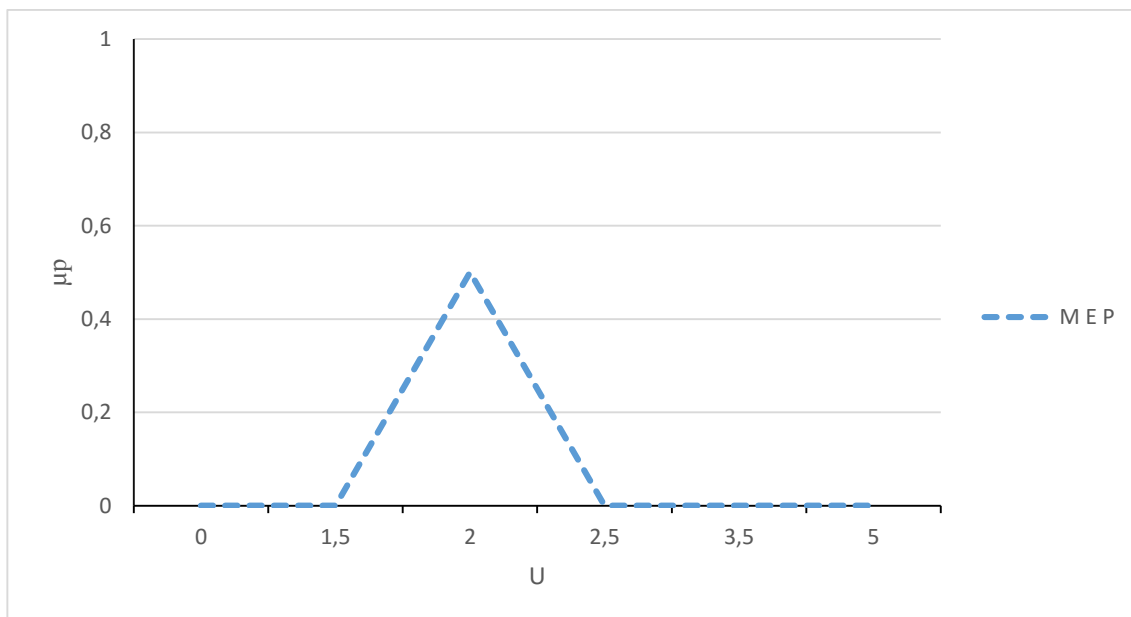
Associatividade: $S(C, S(D, E)) = S(S(C, D), E)$

Monotonicidade: se $C \leq D$ e $E \leq F$, então $S(C, E) \leq S(D, F)$

Coerência nos contornos: $S(C, 1) = a$ e $S(C, 0) = 0$

Dessa forma a intersecção corresponde sempre ao conectivo “E”, representado na figura 3.9 com o conjunto “médio e pequeno”, utilizando como norma a função mínimo.

Figura 3.9 – Representação do conjunto Médio E Pequeno ($M \cap P$)



Fonte: (REZENDE, 2006)

3.2 Ferramentas e Aplicações

Entre 1970 e 1980 diversas aplicações industriais da lógica nebulosa tiveram maior destaque na Europa, após 1980 as primeiras aplicações no Japão foram em um tratamento de água e em um sistema de metrô e finalmente por volta de 1990 empresas dos Estados Unidos despertaram maior interesse nas aplicações. (ABAR, 2015).

Na imagem 3.10 é possível visualizar um quadro com as principais ferramentas e suas características. (ARRUDA; ABUD; PONTES; PONTES; OLIVEIRA, 2013).

Figura 3.10 – Principais ferramentas e suas características

Ferramenta	Site do Projeto	Distribuição	Linguagem	FCL	Última Versão
AForge.NET	http://www.aforgenet.com	GNU LGPL	C#	NO	2012
DotFuzzy	http://www.havana7.com/dotfuzzy/default.aspx	GNU LGPL	C#	XML	2009
Flip++	http://www.dbai.tuwien.ac.at/proj/StarFLIP/	GNU GPL	C++	NO	2005
Fool & Fox	http://rhaug.de/fool/	GNU GPL	C/Java	NO	2002
Free Fuzzy Logic Library	http://fll.sourceforge.net	BSD 3	C/C++	IEC	2003
Fuzzy Inference Engine	http://people.clarkson.edu/~esazonov/FuzzyEngine.htm	GNU GPL	Java	NO	2005
Fuzzy Logic Tools	http://uhu.es/antonio.barragan/category/temas/fuzzy-logic-tools	GNU GPL	C++	NO	2012
Fuzzy Logic Tool Box - MatLab®	http://www.mathworks.com/products/fuzzylogic/	Software Proprietário	Linguagem residente do ambiente MatLab.	Própria	2012
FuzzyClips	http://www.ortech-engr.com/fuzzy/fzycips.html	Software Proprietário ⁹ .	CLIPS	NO	2004
Fuzzy-Lite	http://www.fuzzylite.com	Apache 2.0	C++	IEC ou padrão MatLab	2011
FuzzyTech®	http://www.fuzzytech.com/	Software Proprietário	Não Informado	IEC	2012
jFuzzy Logic	http://jfuzzylogic.sourceforge.net/html/index.html	GNU LGPL	Java	IEC	2012
jFuzzy Qt	http://jfuzzyqt.sourceforge.net/	GNU GPL	C++	IEC	2011
mbFuzzIT	http://mbfuzzit.sourceforge.net/en/mbfuzzit_software.html	GNU GPL	Java	NO	2005
Fuzzy Logic Toolkit - Octave	http://octave.sourceforge.net/index.html	GNU GPL	Linguagem residente do ambiente Octave.	padrão MatLab	2012
Pyfuzzy	http://pyfuzzy.sourceforge.net/	GNU LGPL	Python	IEC	2009
sciFLT	http://www.geocities.ws/jaime_urzua/sciFLT/sciflt.html	GNU GPL	C/Fortran/TC L/SCILAB	padrão MatLab	2004
XFuzzy	http://www2.imse-cnm.csic.es/Xfuzzy/	GNU GPL até a versão 3.0 e BSD 3 na versão 3.3	Java	XFL3	2012

Fonte: (ARRUDA, ABUD, PONTES, PONTES, OLIVEIRA, 2013)

4 LÓGICA NEBULOSA APLICADO A COMPILADORES

Apresentados os principais conceitos de compiladores e Lógica nebulosa, agora é possível entender as diferentes etapas onde a lógica nebulosa pode ser aplicada com o intuito de tolerar falhas léxicas e sintáticas.

Na análise léxica, com a lógica nebulosa, aplica-se o conceito de reconhecimento parcial de cadeias enquanto que na análise sintática aceita-se erros e efetua a recuperação dessas inconformidades, desde que estejam definidas na gramática. (MACIEL, 2006). (CARVALHO; OLIVEIRA; HENRIQUES, 2014)

4.1 Análise Léxica com conceitos *fuzzy*

Assim como na análise léxica utiliza-se de um autômato finito, na análise léxica *fuzzy* utiliza-se de um autômato finito *fuzzy*.

4.1.1 Autômato Finito *Fuzzy*

O autômato finito tem um papel importante no reconhecimento de cadeias e também pode ser aplicado ao reconhecimento aproximado de cadeias, porém seu custo computacional pode tornar inviável o processamento de uma cadeia mesmo tolerando uma quantidade pequena de erros. (MACIEL, 2006).

Uma forma de diminuir esse custo computacional é através da utilização de um autômato finito nebuloso (AFN). Um AFN é um modelo derivado do autômato finito, que possui a capacidade de realizar um processamento e uma validação menos limitada, com uma faixa de valores de pertinência que representa os erros encontrados durante o processamento. (MACIEL, 2006).

Um autômato finito nebuloso é composto por uma quintupla $AFN = (Q, \epsilon, \mu, S, F)$ onde: (MACIEL, 2006)

Q conjunto finito de estados

ϵ é o conjunto de símbolos ou alfabeto

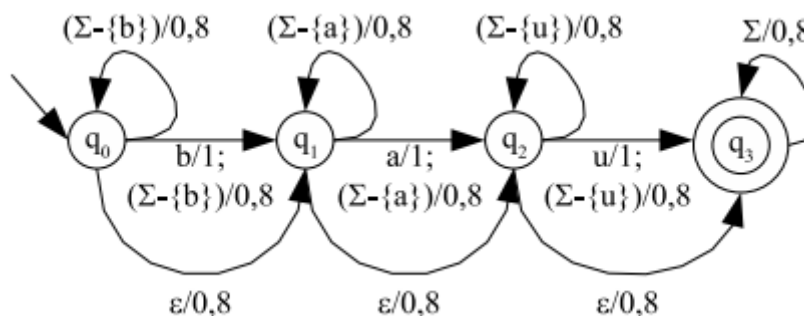
μ é a relação de transição nebulosa

S conjunto nebuloso de estados iniciais

F conjunto nebuloso de estados finais

Existem diversas definições que variam de acordo com o intuito do autor ou com a aplicação do autômato em si, na figura 4.1 é demonstrado um AFN capaz de realizar o reconhecimento aproximado da cadeia “bau”. (MACIEL, 2006).

Figura 4.1 – AFN para palavra bau



Fonte: (MACIEL, 2006)

Com as transições demonstrada na figura 4.1 é possível representar as operações de inclusão, exclusão e substituição de um caractere em uma cadeia. (MACIEL, 2006).

A figura 4.2 demonstra passo a passo do processamento da cadeia “aa” no autômato descrito acima.

Figura 4.2 – Processamento de um AFN da palavra “aa”

	Símbolo consumido	μq_0	μq_1	μq_2	μq_3
Inicial		1	0	0	0
Passo1	vazio	1	0,8	0,64	0,512
Passo2	a	0,8	0,8	0,8	0,512
Passo3	vazio	0,8	0,8	0,8	0,64
Passo4	a	0,64	0,64	0,8	0,64
Passo5	vazio	0,64	0,64	0,8	0,64

Ainda na figura 4.2 é possível entender o algoritmo de validação, onde inicia-se o processamento no estado inicial com pertinência total (1), em seguida, alterna-se entre o consumo das transições de símbolo vazio e o símbolo disponível na fita. Ao término do processamento conclui-se que a quantidade mínima de operações necessárias para transformar “aa” em “bau” é uma substituição e uma inclusão com 0,64 de pertinência na validação. (MACIEL, 2006).

4.2 Análise Sintática com conceitos *fuzzy*

Assim como na análise sintática utiliza-se de uma gramática, na análise sintática *fuzzy* utiliza-se de uma gramática *fuzzy*.

4.2.1 Gramática *Fuzzy* Livre de Contexto

Embora existam diversas definições, um dos primeiros trabalhos proposto por Hahn (HAHN, 1989) representa uma gramática *fuzzy* livre de contexto sendo uma quádrupla $FG = (V_n, V_t, P, s)$, onde:

1. V_n é o conjunto de símbolos não terminais
2. V_t é o conjunto de símbolo terminais
3. P é o conjunto de regras de produções
4. s é o símbolo inicial

E cada regra de produção composto por $A \rightarrow \beta, \alpha \mid 0 \leq \alpha \leq 1$, onde:

- A é um símbolo não terminal
- β é um conjunto de terminais ou não terminais
- α é a pertinência da regra que será propagada caso seja consumida

Na figura 4.3 é observa-se um exemplo de uma gramática *fuzzy*.

Figura 4.3 – Exemplo de uma gramática *fuzzy*

$$\begin{aligned}
 FG &= (Vn, Vt, P, s) \\
 Vn &= \{S, A\} \\
 Vt &= \{a, b\} \\
 P &= \{ S \rightarrow bbA, 1 \\
 &\quad A \rightarrow aaS, 1 \\
 &\quad A \rightarrow aS, 0.9 \}
 \end{aligned}$$

Ainda na figura 4.3 é possível observar as diferentes pertinências (1, 1 e 0.9) nas distintas regras da gramática.

Neste trabalho implementou-se uma versão modificada do algoritmo LR(1) proposto por (KOPPLER, 1999), capaz de propagar as pertinências de cada regra, conforme demonstrado na figura 4.4.

Figura 4.4 – Algoritmo LR(1) modificado

```

1  pertinence = 1;
2  while(1) { /*repita indefinidamente*/
3      seja s o estado no topo da pilha;
4      if(ACTION[s,a] = shift t)
5      {
6          empilha t na pilha;
7          seja a o próximo símbolo da entrada;
8      }else if(ACTION[s,a] = reduce A->B, x)
9      {
10         desempilha símbolos |B| da pilha;
11         faça estado t agora ser o topo da pilha;
12         empilhe GOTO[t,A] na pilha;
13         imprima a produção A -> B, x;
14         pertinence = norm(pertinence, x);
15     }else if(ACTION[s,a] = accept) pare; /*Análise terminou*/
16     else raise erro;
17 }

```

4.2.2 Gramática Livre de Contexto Probabilística

Como uma outra forma de abordar a correção sintática, existe a gramática probabilística (GLCP), onde são adicionados probabilidade no consumo de cada regra. (ZADROZNY, 2015).

Também conhecida como gramática estocástica a GLCP é composta por uma quadrupla $GLCP = (V_n, V_t, P_s, S)$, onde (OLIVEIRA; FERRAMOLA, 2015):

1. V_n é o conjunto de símbolos não terminais
2. V_t é o conjunto de símbolo terminais
3. P_s é o conjunto de produções estocásticas
4. S é o símbolo inicial

E cada regra de produção composto por $A \rightarrow \beta, p \mid 0 \leq p \leq 1$, onde:

- A é um símbolo não terminal
- β é um conjunto de terminais ou não terminais
- p é a probabilidade da regra

Na figura 4.5 é observa-se um exemplo de uma gramática probabilística.

Figura 4.5 – Exemplo de uma gramática probabilística.

$$\begin{aligned}
 GLCP &= (V_n, V_t, PS, S) \\
 V_n &= \{S, A, B\} \\
 V_t &= \{0, 1\} \\
 PS &= \{ S \rightarrow 1A, 1 \\
 &\quad A \rightarrow 0B, 0.8 \\
 &\quad A \rightarrow 1, 0.2 \\
 &\quad B \rightarrow 0, 0.3 \\
 &\quad B \rightarrow 1S, 0.7 \}
 \end{aligned}$$

Fonte: (TSURUOKA; TSUJII, 2015)

Elas são largamente utilizadas no processamento de linguagens naturais e possuem algoritmos específicos para processamento como o VITERBI e o CYK,

embora ambos exigem que a gramática esteja na forma normal de CHOMSKY, o segundo é mais utilizado por ser mais eficiente (TSURUOKA; TSUJII, 2015).

Na figura 4.6 é possível ver o algoritmo CYK com a recuperação da árvore sintática.

Figura 4.6 – Algoritmo LR(1) modificado

```

1  N = length(sentence);
2  for (i = 1 to N) {
3      word = sentence[i];
4      for (each rule "POS --> Word [prob]" in the grammar)
5          P[POS,i,i] = new Tree(POS,i,i,word,null,null,prob);
6      }
7
8  for (length = 2 to N)          % length = length of phrase
9      for (i = 1 to N+1-length) { % i == start of phrase
10         j = i+length-1;         % j == end of phrase
11         for (each NonTerm M) {
12             P[M,i,j] = new Tree(M,i,j,null,null,null,0.0);
13             for (k = i to j-1)   % k = end of first subphrase
14                 for (each rule "M -> Y,Z [prob]" in the grammar) {
15                     newProb = P[Y,i,k].prob * P[Z,k+1,j].prob * prob;
16                     if (newProb > P[M,i,j].prob) {
17                         P[M,i,j].left = P[Y,i,k];
18                         P[M,i,j].right = P[Z,k+1,j];
19                         P[M,i,j].prob = newProb;
20                     } % endif line 15
21                 } % endfor line 13
22             } % endfor line 10
23         } % endfor line 8
24
25     return P;
26 } % end CYK-PARSE.

```

Fonte: (DAVIS, 2015)

É importante entender que conceitualmente a probabilidade associada a cada etapa de derivação é igual ao produto das probabilidades associadas na sequência de cada regra estocástica, enquanto que na gramática *fuzzy* a pertinência de cada derivação é o resultado da operação de norma entre as pertinências das regras utilizadas. (OLIVEIRA; FERRAMOLA, 2015).

5 FRAMEWORK PARA CONSTRUÇÃO DE COMPILADORES COM CONCEITOS FUZZY

Um *framework* é um conjunto de ferramentas e bibliotecas que auxiliam no desenvolvimento de um software, com os conceitos apresentados nos capítulos anteriores desenvolveu-se um conjunto de funcionalidades que ajudam na definição e elaboração de um compilador que utilize de conceitos *fuzzy*. Nos próximos tópicos serão abordadas as diferentes etapas da ferramenta desenvolvida.

5.1 Autômato e REGEX *Fuzzy*

Primeiramente implementou-se um autômato *fuzzy* e um conjunto de classes capaz de representar e processar uma REGEX *fuzzy* conforme demonstra a figura 5.1.

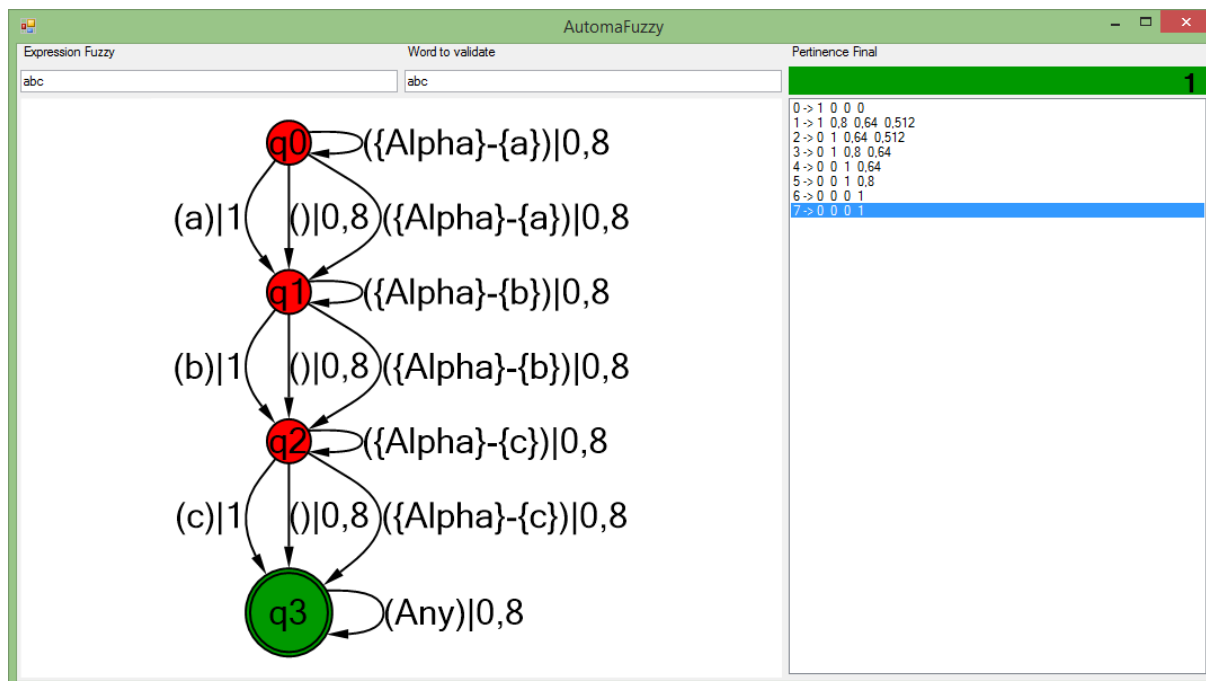
Figura 5.1 – Diagrama de classe módulo de Autômato *Fuzzy* e REGEX *Fuzzy*

Enquanto que as classes de State, Automa e Transition representam os principais componentes de um autômato *fuzzy*, a Classe AbstractRule é responsável por determinar cada tipo de regra de transição do autômato. Com as classes herdadas de AbstractRule, presentes no diagrama, é possível implementar as operações básica de inserção, remoção e substituição de caracteres.

Existe a possibilidade de utilizar o autômato desenvolvido para outras aplicações além do reconhecimento aproximado de cadeias, e novas operações, para processamento de caracteres, podem ser implementadas a partir de uma herança da classe AbstractRule.

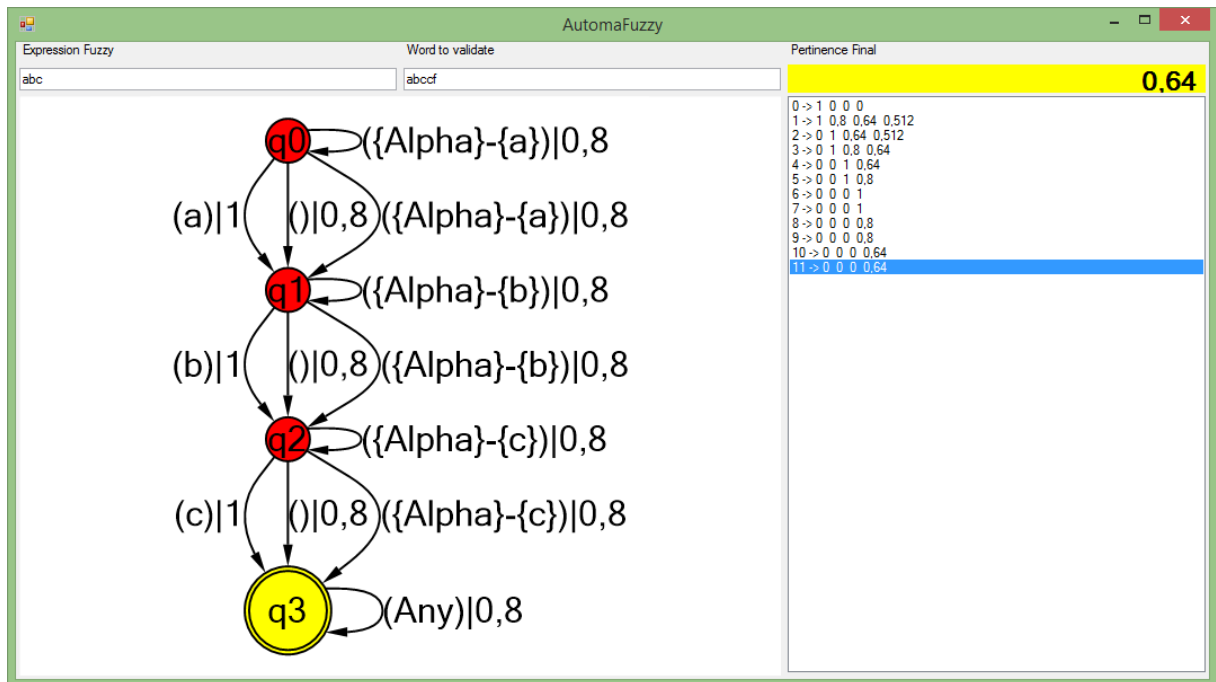
Para o correto entendimento e comprovação do funcionamento das classes implementadas, desenvolveu-se uma aplicação para testes, demonstrada na figura 5.2.

Figura 5.2 – Aplicação de testes do módulo de autômato *fuzzy*



Ainda na imagem 5.2 é possível observar o autômato *fuzzy*, correspondente a expressão regular *fuzzy* “abc” e as diferentes pertinências para cada transição. No lado direito da figura é demonstrado o passo-a-passo da validação da cadeia “abc”, em destaque a pertinência total “1” na validação e o último passo mostrando a pertinência de cada estado.

Na figura 5.3 é demonstrado uma validação parcial, ou seja, uma pertinência de 0,64 da palavra “abccf” sobre a expressão regular *fuzzy* “abc”.

Figura 5.3 – Pertinência parcial aplicação de testes do módulo de autômato *fuzzy*

Para utilizar o módulo de REGEX *fuzzy* basta instanciar a classe `RecognitionFuzzy` e após isso é possível validar qualquer sequência de caracteres utilizando o método `Match`, que retorna a pertinência obtida, conforme demonstrado na figura 5.4.

Figura 5.4 – Exemplo utilização Classe `RecognitionFuzzy`

```
RecognitionFuzzy regexFuzzy = new RecognitionFuzzy("[0-9]+");
double pertinence = regexFuzzy.Match("123456a");
```

Ao término da execução a variável “`pertinence`” receberá o valor “0,8”, devido a presença do caractere “a” e as pertinências das transições, conforme o processamento do autômato.

Ainda com o mesmo exemplo é possível definir diferentes Normas e Conormas para o processamento do autômato, alterando assim o resultado do mesmo processamento, conforme demonstrado na figura 5.5.

Figura 5.5 – Exemplo utilização com definição de norma e conorma

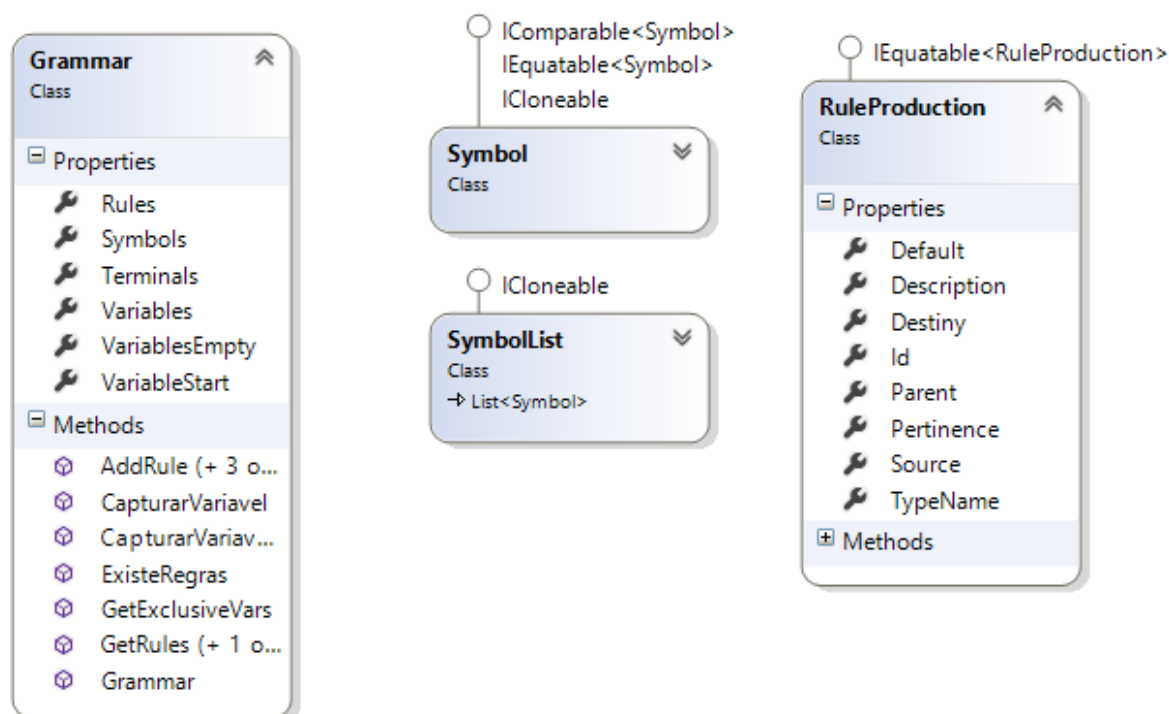
```
RecognitionFuzzy regexFuzzy = new RecognitionFuzzy("[0-9]+",
    new MinNorm(), new MaxConorm(), 0.3);
double pertinence = regexFuzzy.Match("123456a");
```

Além das alterações de norma e conorma, ainda na figura 5.5, com o valor “0,3” as transições do autômato fuzzy foram redefinidas para essa pertinência, ao término do processamento a variável “pertinence” receberá o valor “0,3”.

5.2 Gramática Fuzzy

Nessa etapa do desenvolvimento, implementou-se as classes que representam uma gramática *fuzzy* conforme diagrama demonstrado na figura 5.6.

Figura 5.6 – Diagrama de classe módulo Gramática Fuzzy



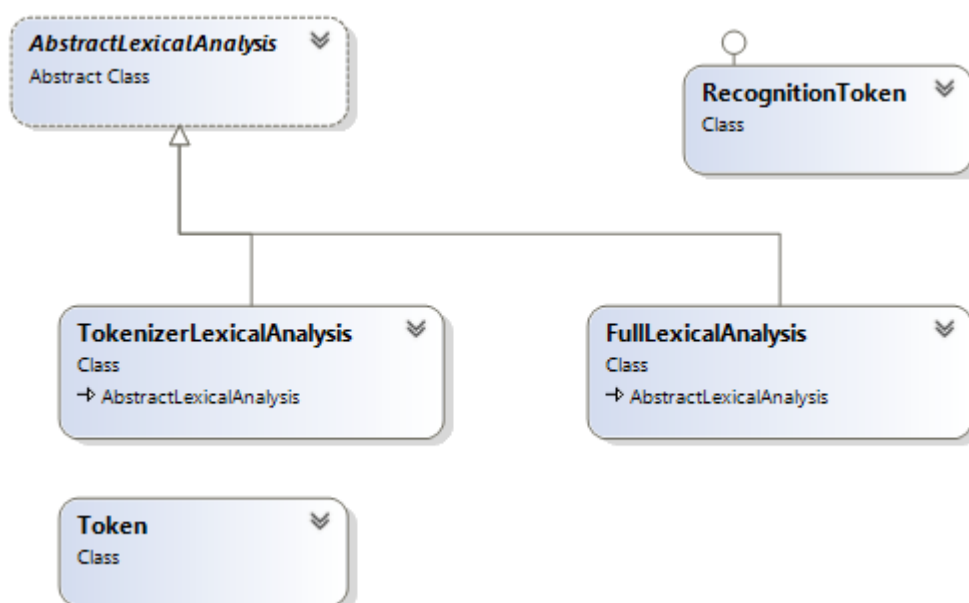
A partir das classes Grammar e RuleProduction é possível a representação de uma gramática nebulosa. Com a classe Symbol representa-se os símbolos terminais

e não terminais, enquanto que a classes `SymbolList` representa a sequência de símbolos de cada regra.

5.3 Análise Léxica *Fuzzy*

Com as classes necessárias para a análise léxica definidas, implementou-se as classes para o processamento do autômato, conforme demonstrado na figura 5.7.

Figura 5.7 – Diagrama de classe módulo Análise Léxica *Fuzzy*



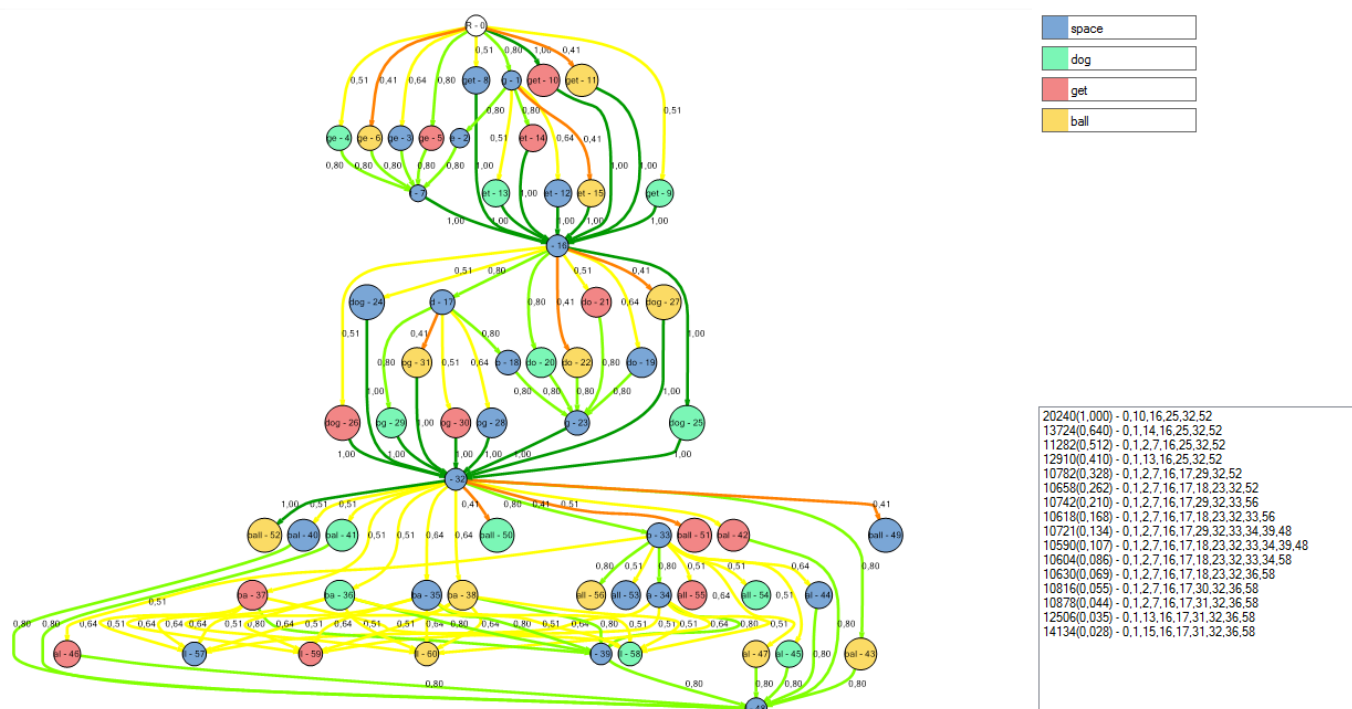
Com as classes `RecognitionToken` e `Token` é possível representar cada *token* presente na gramática e cada conjunto de tipo e valor reconhecido no código de origem a ser processado, respectivamente.

Além dessas classes, na figura 5.7 ainda se observa as classes `FullLexicalAnalysis` e `TokenizerLexicalAnalysis`, que são dois modos disponibilizados para a realização da análise léxica, enquanto uma tenta realizar o match das REGEX *fuzzy* em todas as sub-cadeias dentro do código de origem, a outra realiza a quebra por espaço, e para cada item da lista tenta realizar o match.

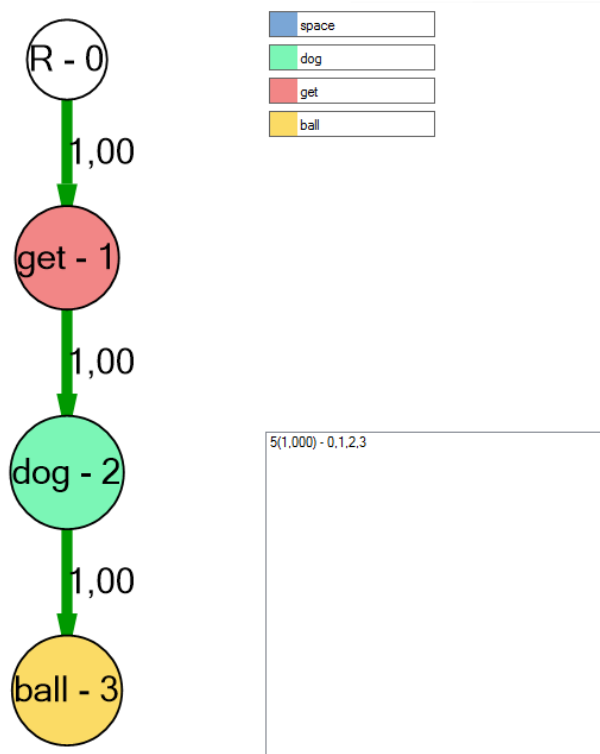
Nas figuras 5.8 e 5.9 é possível comparar as diferenças entre os tipos de análise léxica para o processamento da cadeia “dog get ball” com os *tokens* “dog”, “get” e “ball” com REGEX *fuzzy* “dog”, “get” e “ball”, respectivamente. Nos arcos estão

as pertinências de cada cadeia analisada com a REGEX *fuzzy* correspondente ao *token*.

Figura 5.8 – Análise Léxica sem a técnica de Tokenizer



Na figura 5.9 demonstra o mesmo processamento utilizando a técnica de *Tokenizer*.

Figura 5.9 – Análise Léxica utilizando a técnica de *Tokenizer*

Na figura 5.9 observa-se a diminuição significativa dos números de estados e de arcos, pois foram processadas apenas sub-cadeias presente na cadeia “dog get ball” obtidas pela separação por espaço (“dog”, “get” e “ball”), e ainda houve apenas o processamento das maiores pertinências encontradas.

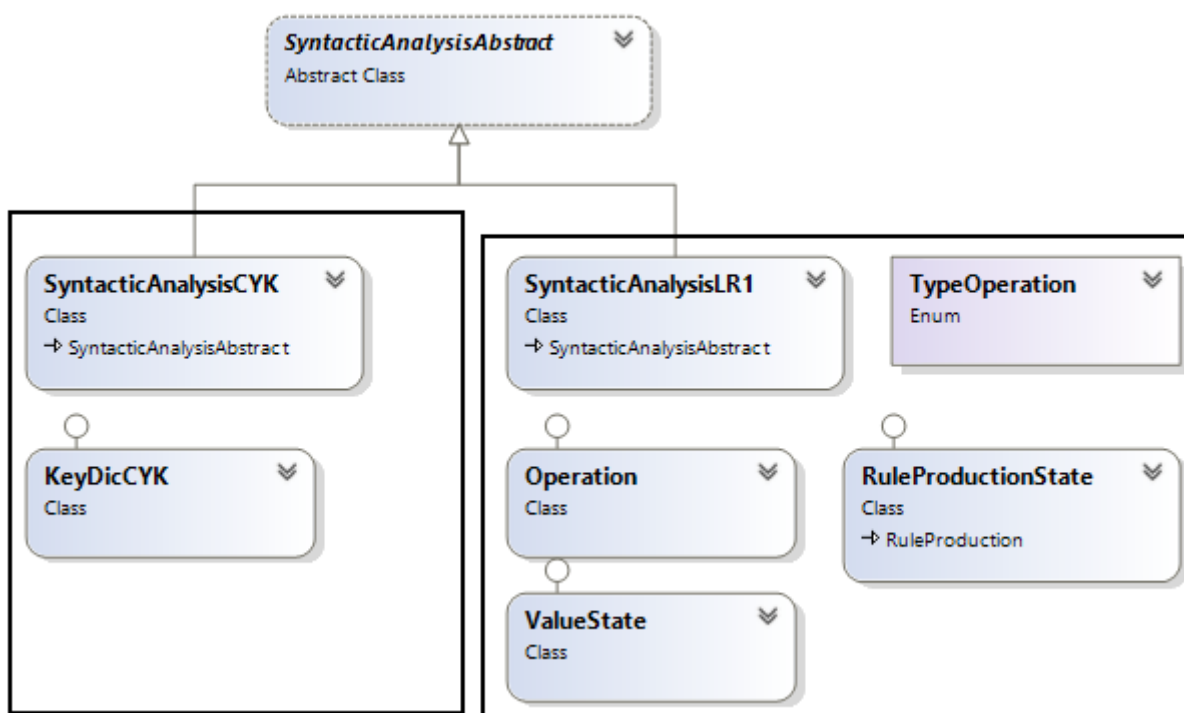
As cores vermelho, verde e amarelo representam os *tokens* encontrados “get”, “dog” e “ball” ambos com pertinência total (1), devido a correta grafia. O fluxo nas figuras 5.8 e 5.9 são apenas representativos dos caminhos que análise léxica fuzzy pode gerar.

É importante lembrar que são validados *token a token*, sem ser realizado qualquer validação da ordem dos *tokens*, após o processamento é gerado uma lista com os *tokens* reconhecidos para ser propagado para próxima etapa, a análise sintática.

5.4 Análise Sintática *Fuzzy*

Utilizando as classes do módulo de Gramática *Fuzzy* definiu-se as classes para a validação e geração de árvore sintática, conforme demonstra a figura 5.9.

Figura 5.10 – Análise Léxica utilizando a técnica de Tokenizer

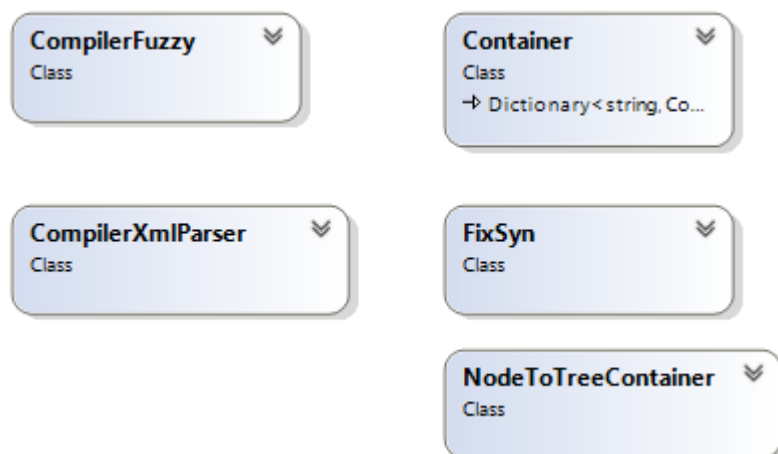


Na figura 5.10 é possível observar que, através da herança da classe *SyntacticAnalysisAbstract*, as classes *SyntacticAnalysisLR1* e *SyntacticAnalysisCYK* foram implementadas, responsáveis pelo processamento da gramática utilizando o algoritmo LR(1) modificado, capaz de propagar as pertinências das regras, e o analisador CYK respectivamente.

5.5 Compilador *Fuzzy*

Finalmente com os analisadores definidos implementou-se as classes responsáveis por unir cada parte e efetuação da compilação, demonstradas na figura 5.11.

Figura 5.11 – Diagrama de classe módulo compilador



Na figura 5.11 é possível observar as classes `Container`, `FixSyn` e `NodeToTreeContainer`, responsáveis pelas correções da gramática lida para a gramática esperada e geração de uma estrutura mais simples a partir da árvore sintática obtida pelo processamento da análise sintática.

Ainda na figura 5.11 observa-se as classes `CompilerFuzzy` e `CompilerXmlParser`, responsáveis por centralizar todo o processamento e realizar o parser de um xml definido para a estrutura do compilador, respectivamente.

5.6 Exemplo de utilização da ferramenta

Para a realização de testes e comprovação do conceito, novamente foi desenvolvido uma aplicação com interface gráfica demonstrando cada etapa do processamento de uma cadeia dentro do compilador.

Inicialmente desenvolveu-se um arquivo em formato de XML (Extensible Markup Language) para definição da gramática e parametrização do compilador demonstrado na figura 5.12.

Figura 5.12 – XML de Configuração do primeiro Exemplo

```

<?xml version="1.0" encoding="utf-8" ?>
<Compiler>
  <Settings>
    <Norm>MAX</Norm>
    <Conorm>MULTIPLY</Conorm>
    <Parser>SyntacticAnalysisLR1</Parser>
    <Lexer>TokenizerLexicalAnalysis</Lexer>
  </Settings>
  <RecTokens>
    <RecToken id="0" name="space" regex=" " color="#7aa6d6"/>
    <RecToken id="1" name="dog" regex="dog" color="#7bf6b6"/>
    <RecToken id="3" name="get" regex="get" color="#f28686"/>
    <RecToken id="5" name="ball" regex="ball" color="#fbd665"/>
  </RecTokens>
  <Grammar norm="Multiply" conorm="Max">
    <Symbols>
      <!-- Terminals -->
      <Symbol id="1" name="dog" terminal="true" recTokenId="1" charValue=""/>
      <Symbol id="3" name="get" terminal="true" recTokenId="3" charValue=""/>
      <Symbol id="5" name="ball" terminal="true" recTokenId="5" charValue=""/>

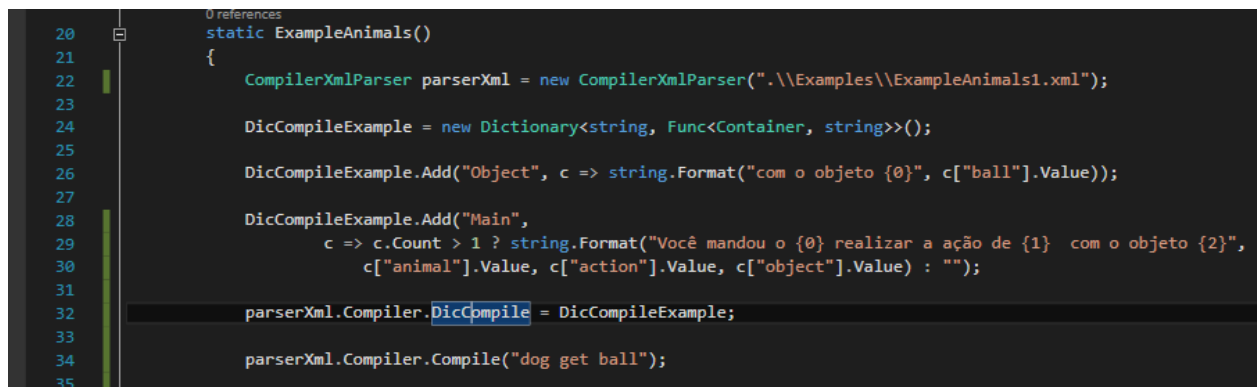
      <!-- Variables -->
      <Symbol id="9" name="Initial" terminal="false" recTokenId="" charValue="" variableInitial="true"/>
      <Symbol id="10" name="Other" terminal="false" recTokenId="" charValue="" variableInitial="false"/>
    </Symbols>
    <Rules>
      <Rule id="1" typeName="Main" sourceName="Initial" pertinence="1" idRuleParent="" default="true">
        <Description><![CDATA[Initial=>dog get ball]]></Description>
        <Destinys>
          <Destiny name="dog" var="animal"/>
          <Destiny name="get" var="action"/>
          <Destiny name="ball" var="object"/>
        </Destinys>
      </Rule>
      <Rule id="2" typeName="Main" sourceName="Initial" pertinence="0.1" idRuleParent="1"
        default="false">
        <Description><![CDATA[Initial=>get]]></Description>
        <Destinys>
          <Destiny name="get" var="action"/>
        </Destinys>
      </Rule>
      <Rule id="3" typeName="Main" sourceName="Initial" pertinence="0.9" idRuleParent="1"
        default="false">
        <Description><![CDATA[Initial=>get dog ball]]></Description>
        <Destinys>
          <Destiny name="get" var="action"/>
          <Destiny name="dog" var="animal"/>
          <Destiny name="ball" var="object"/>
        </Destinys>
      </Rule>
    </Rules>
  </Grammar>
</Compiler>

```

Ainda na figura 5.12 é importante notar a configuração inicial na tag “Settings” e a definição das REGEX *fuzzy* e da gramática em si.

Após a definição do XML principal basta importa-lo e a partir da classe `CompilerFuzzy` já é possível o processamento do código fonte, conforme demonstrado na figura 5.13.

Figura 5.13 – Primeiro Exemplo utilização do *Framework*



```

20 0 references
21 static ExampleAnimals()
22 {
23     CompilerXmlParser parserXml = new CompilerXmlParser(".\\Examples\\ExampleAnimals1.xml");
24
25     DicCompileExample = new Dictionary<string, Func<Container, string>>();
26
27     DicCompileExample.Add("Object", c => string.Format("com o objeto {0}", c["ball"].Value));
28
29     DicCompileExample.Add("Main",
30         c => c.Count > 1 ? string.Format("Você mandou o {0} realizar a ação de {1} com o objeto {2}",
31             c["animal"].Value, c["action"].Value, c["object"].Value) : "");
32
33     parserXml.Compiler.DicCompile = DicCompileExample;
34
35     parserXml.Compiler.Compile("dog get ball");

```

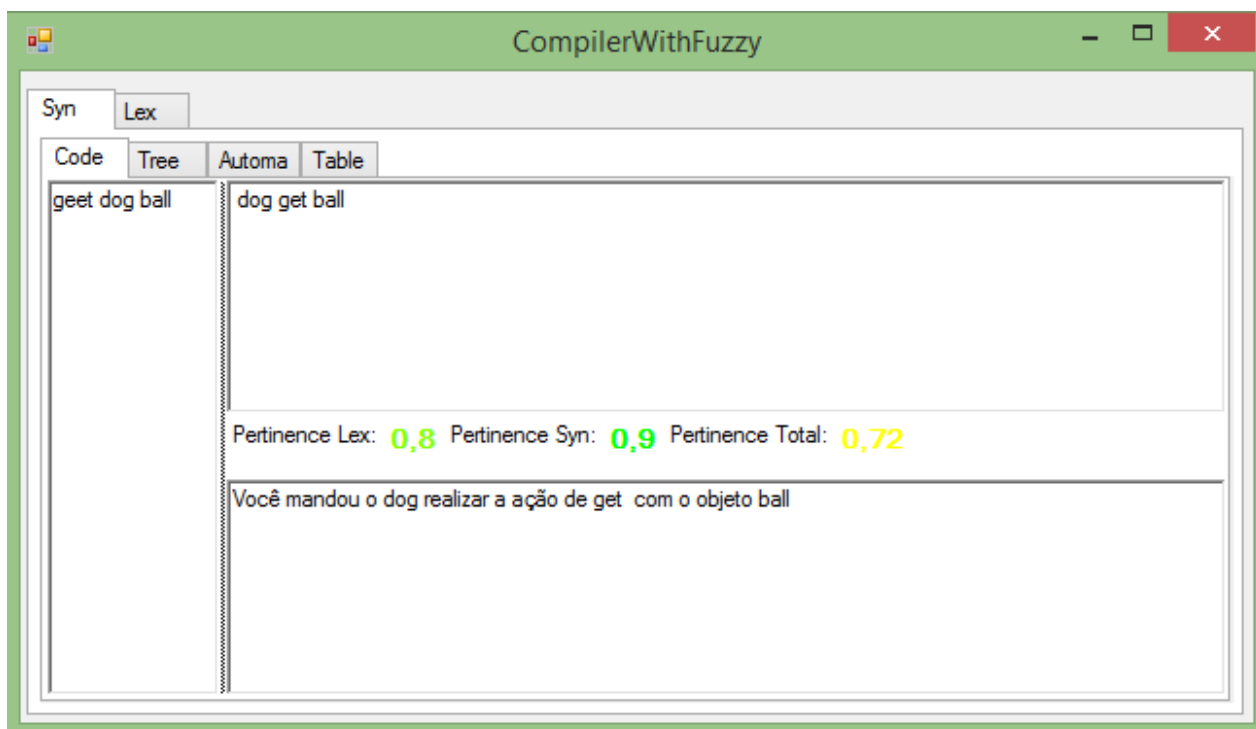
Ainda na figura 5.13, a partir da linha 24 até a linha 32, é possível observar a definição dos métodos responsáveis por realizar o *parser* de cada Type(definido no XML) encontrado no código fonte para processamento.

Finalmente após essa definição foram desenvolvidas as interfaces que torna possível a visualização de cada etapa do compilador.

A figura 5.14 demonstra a tela inicial, com a cadeia a ser processada (lado esquerdo) a cadeia lida e corrigida (lado superior direito) e a cadeia resultante compilada (lado inferior direito).

É importante notar que a cadeia a ser processada (“geet dog ball”) possui erros tantos léxicos (“geet” deveria ser get) quanto sintático (“geet dog ball” deveria ser “dog get ball”).

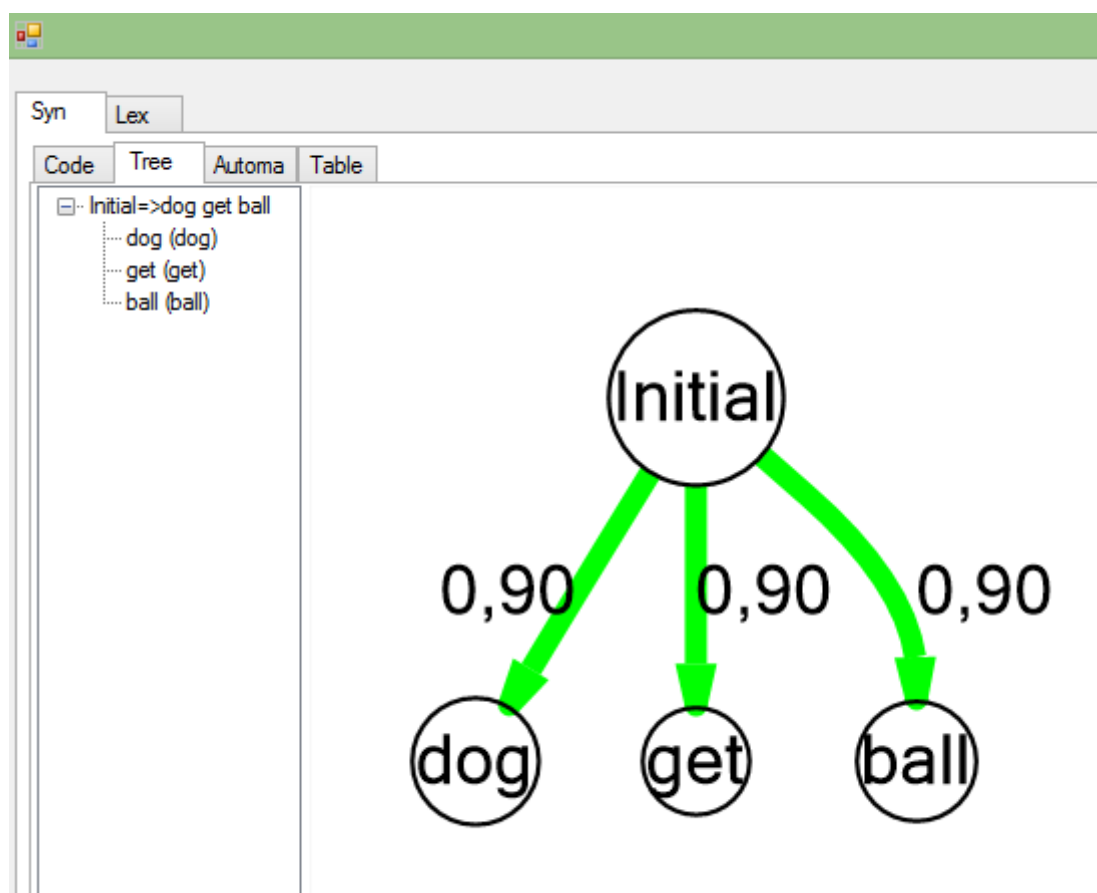
Figura 5.14 – Tela inicial processamento



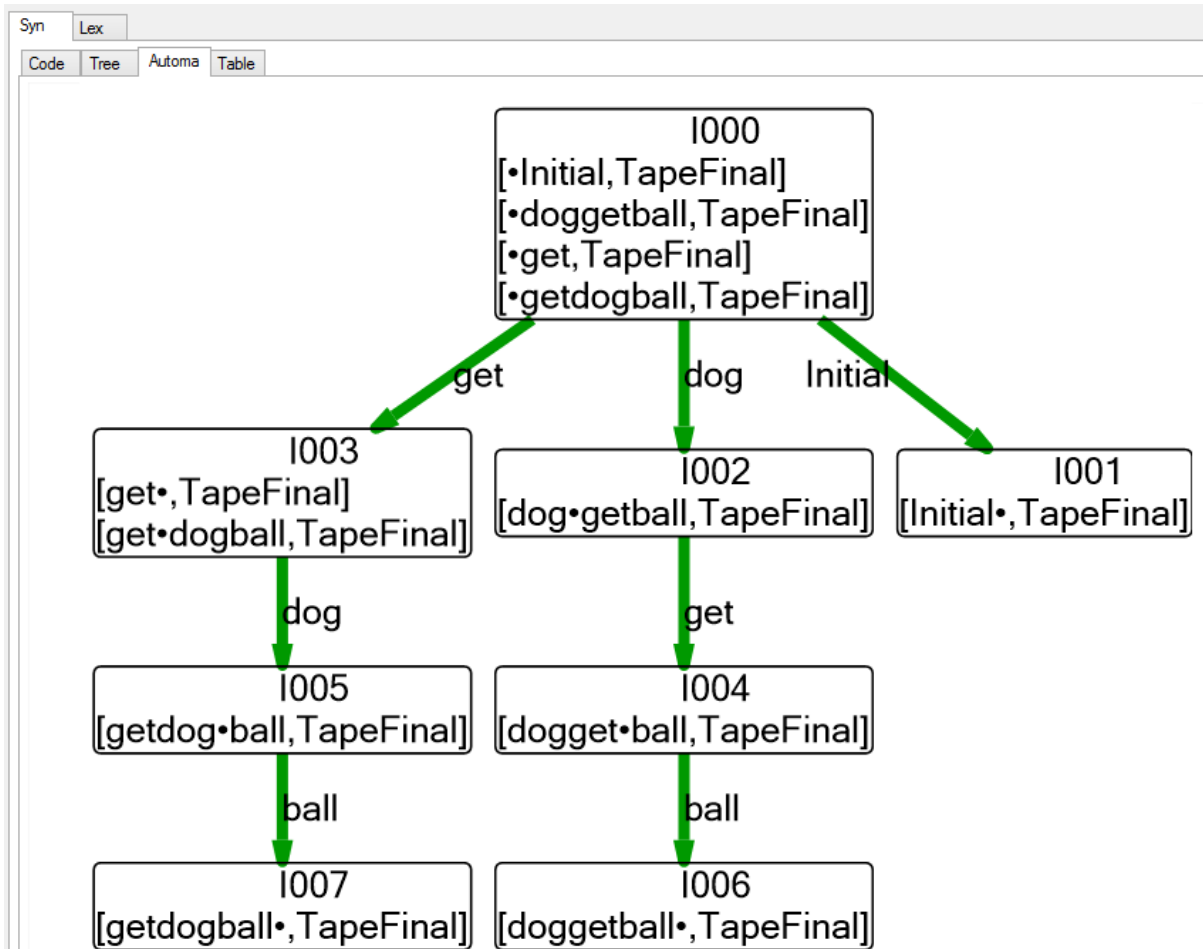
Ainda na figura 5.14 é possível notar a pertinência Léxica de 0.8 indicando o erro de “geet” para “get”, a pertinência sintática de “0.9” definida na regra 3 da imagem 5.12 e a pertinência total, sendo esta última obtida a partir da operação da norma “multiplicação” entre a pertinência Léxica e Sintática.

Na figura 5.15 observa-se a árvore sintática gerada após o processamento do analisador sintático, é importante notar a pertinência de “0,9” devido a utilização da regra definida no XML.

Figura 5.15 – Árvore sintática gerada (corrigida) a partir do processamento

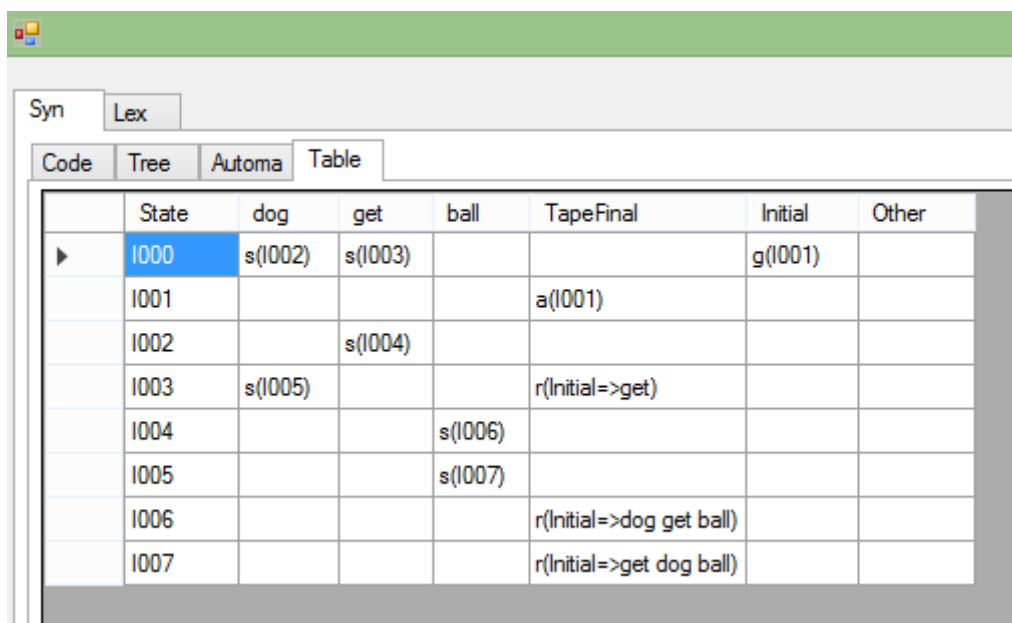


A figura 5.16 demonstra o autômato gerado do algoritmo LR(1) para o processamento.

Figura 5.16 – Autômato intermediário da análise sintática LR(1) *fuzzy*

A figura 5.17 demonstra a tabela de transições e operações para a correta validação de um texto no algoritmo LR(1).

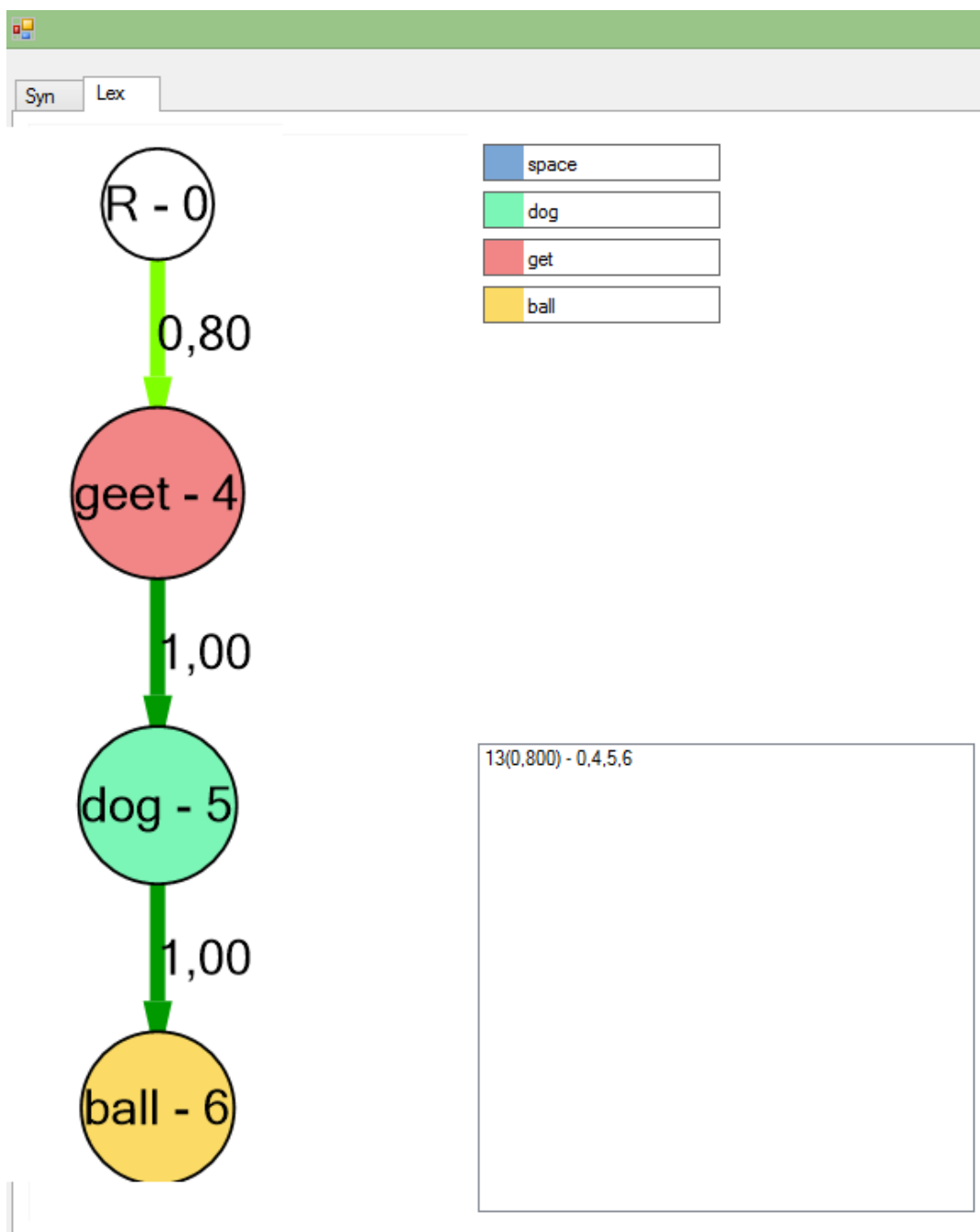
Figura 5.17 – Tabela de transições e operações algoritmo LR(1)



	State	dog	get	ball	TapeFinal	Initial	Other
►	1000	s(1002)	s(1003)			g(1001)	
	1001				a(1001)		
	1002		s(1004)				
	1003	s(1005)			r(Initial=>get)		
	1004			s(1006)			
	1005			s(1007)			
	1006				r(Initial=>dog get ball)		
	1007				r(Initial=>get dog ball)		

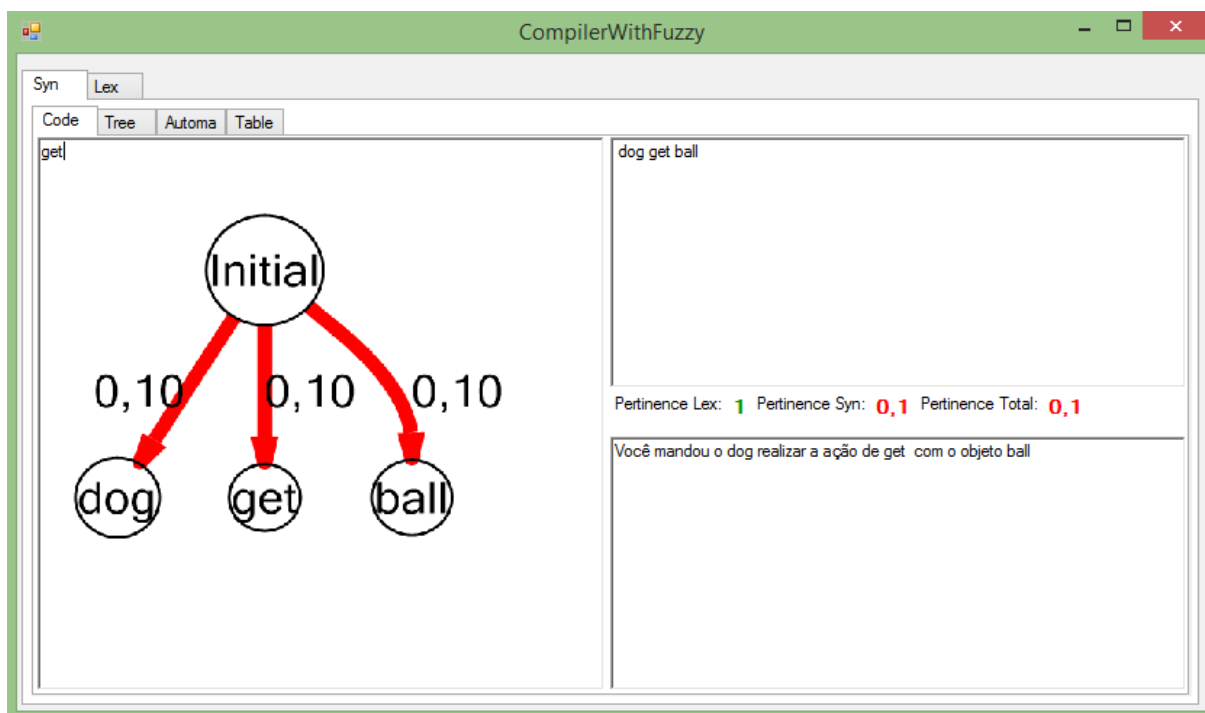
Enquanto que as imagens anteriores demonstram as etapas da correção sintática, a figura 5.18 demonstra a correção léxica, onde a pertinência 0.8 na palavra “geet” representa o custo de uma remoção de um caractere para se transformar no token “get”.

Figura 5.18 – Tela de demonstração do processamento léxico



Entendido um exemplo inicial é possível aumentar as definições de regras e de diferentes valores de pertinência obtendo resultados conforme demonstrado nas figuras 5.19, 5.20 e 5.21.

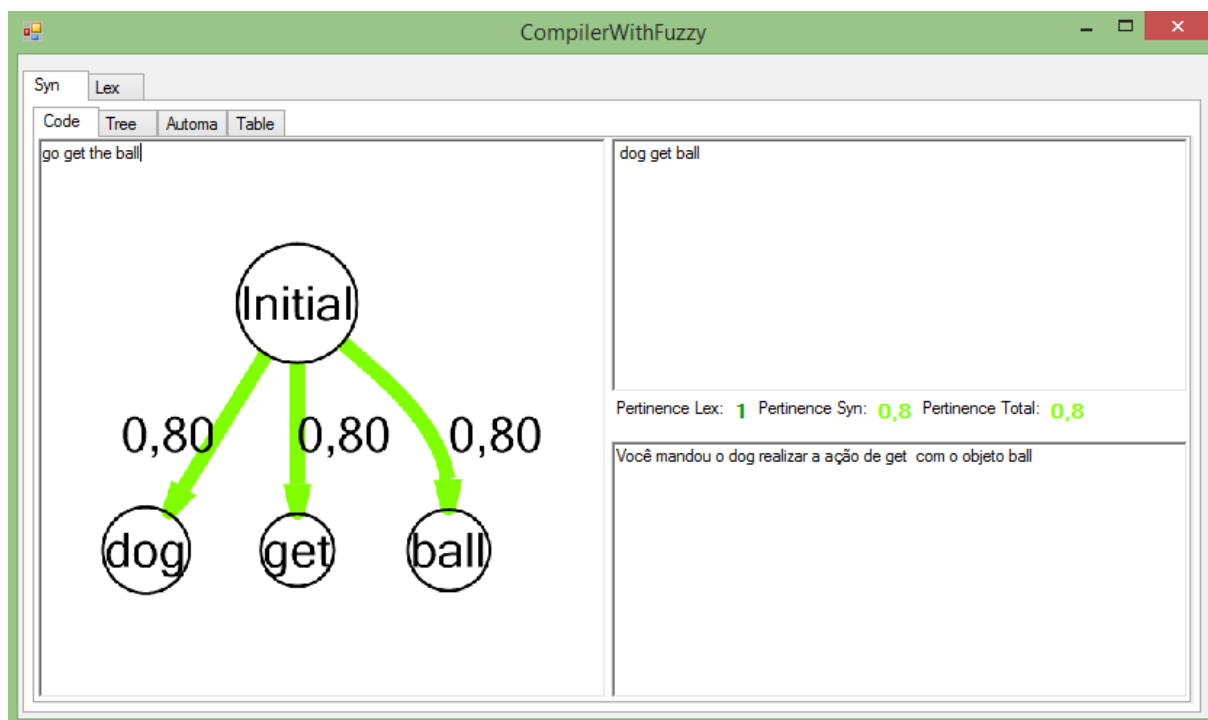
Figura 5.19 – Demonstração de processamento com sentença “get”



Na figura 5.19 é demonstrado a árvore sintática e as diferentes pertinências para o processamento da sentença “get”.

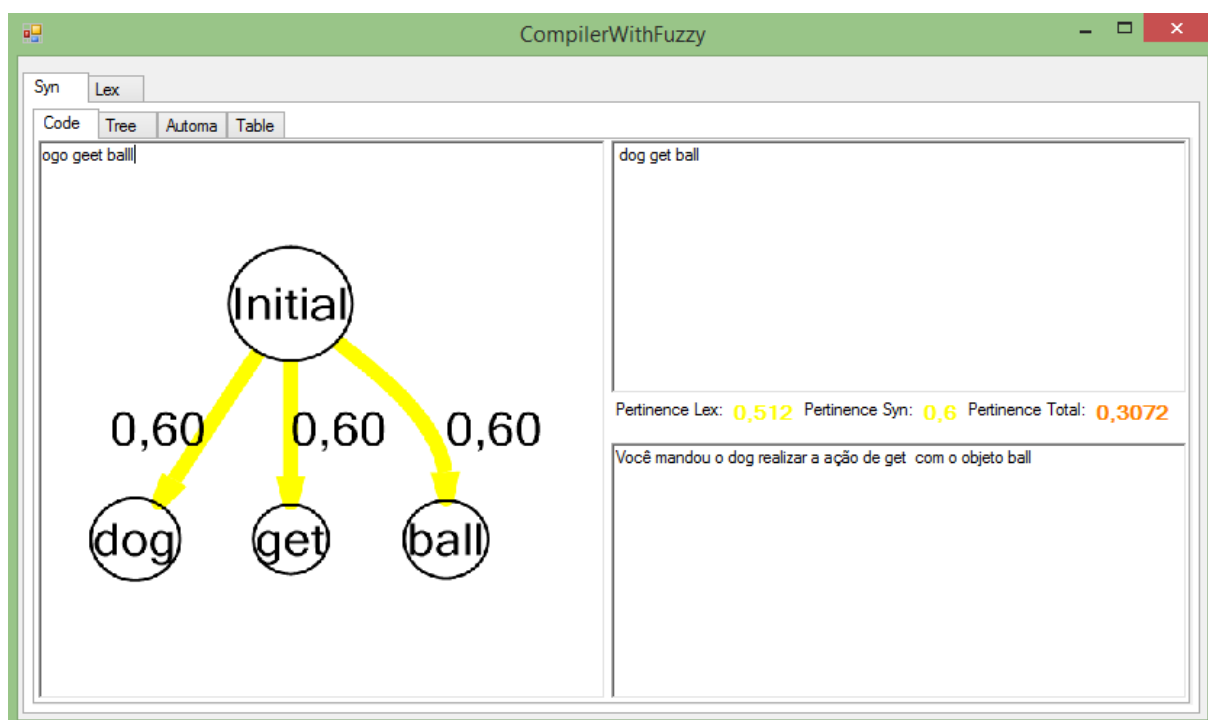
Na figura 5.20 observa-se uma regra adicionada para a sentença “go get the ball”, pertinência de 0,8 e o seu processamento.

Figura 5.20 – Demonstração de processamento com sentença “go get the ball”



Na figura 5.21 é demonstrado o processamento da sentença “ogo geet ball” com diversos erros sintáticos e léxicos, diminuindo a pertinência total para 0,3072.

Figura 5.21 – Demonstração de processamento com sentença “ogo geet ball”



Ainda para efeito de testes definiu-se o XML demonstrado na figura 5.22.

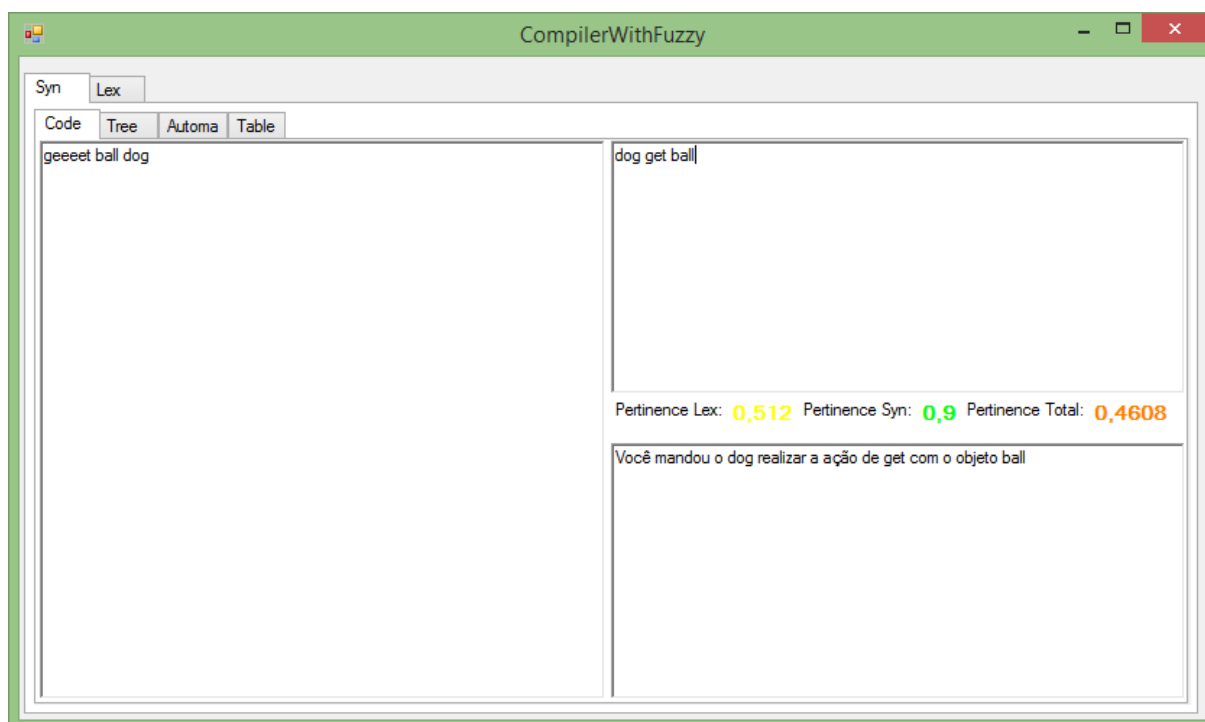
Figura 5.22 – XML de configuração exemplo 2

```
<?xml version="1.0" encoding="utf-8" ?>
<Compiler>
  <Settings>
    <Norm>MAX</Norm>
    <Conorm>MULTIPLY</Conorm>
    <Parser>SyntacticAnalysisCYK</Parser>
    <Lexer>TokenizerLexicalAnalysis</Lexer>
  </Settings>
  <RecTokens>
    <RecToken id="1" name="space" regex=" " color=""/>
    <RecToken id="2" name="dog" regex="dog" color=""/>
    <RecToken id="3" name="get" regex="get" color=""/>
    <RecToken id="4" name="ball" regex="ball" color=""/>
  </RecTokens>
  <Grammar norm="Multiply" conorm="Max">
    <Symbols>
      <!-- Terminals -->
      <Symbol id="1" name="dog" terminal="true" recTokenId="2" charValue=""/>
      <Symbol id="3" name="get" terminal="true" recTokenId="3" charValue=""/>
      <Symbol id="5" name="ball" terminal="true" recTokenId="4" charValue=""/>
      <!-- Variables -->
      <Symbol id="9" name="Initial" terminal="false" recTokenId="" charValue="" variableInitial="true"/>
      <Symbol id="11" name="A" terminal="false" recTokenId="" charValue="" variableInitial="false"/>
      <Symbol id="12" name="B" terminal="false" recTokenId="" charValue="" variableInitial="false"/>
      <Symbol id="13" name="C" terminal="false" recTokenId="" charValue="" variableInitial="false"/>
      <Symbol id="13" name="D" terminal="false" recTokenId="" charValue="" variableInitial="false"/>
    </Symbols>
    <Rules>
      <Rule id="1" typeName="Main" sourceName="Initial" pertinence="1" idRuleParent="" default="true">
        <Description><![CDATA[Initial=>A D]]></Description>
        <Destinys>
          <Destiny name="A" var="a"/>
          <Destiny name="D" var="d"/>
        </Destinys>
      </Rule>
      <Rule id="11" typeName="Main" sourceName="Initial" pertinence="0.9" idRuleParent="" default="true">
        <Description><![CDATA[Initial=>D A]]></Description>
        <Destinys>
          <Destiny name="D" var="d"/>
          <Destiny name="A" var="a"/>
        </Destinys>
      </Rule>
      <Rule id="2" typeName="D" sourceName="D" pertinence="1" idRuleParent="" default="true">
        <Description><![CDATA[D=>B C]]></Description>
        <Destinys>
          <Destiny name="B" var="b"/>
          <Destiny name="C" var="c"/>
        </Destinys>
      </Rule>
      <Rule id="3" typeName="A" sourceName="A" pertinence="1" idRuleParent="" default="true">
        <Description><![CDATA[A=>dog]]></Description>
        <Destinys>
          <Destiny name="dog" var="animal"/>
        </Destinys>
      </Rule>
      <Rule id="4" typeName="B" sourceName="B" pertinence="1" idRuleParent="" default="true">
        <Description><![CDATA[B=>get]]></Description>
        <Destinys>
          <Destiny name="get" var="action"/>
        </Destinys>
      </Rule>
      <Rule id="5" typeName="C" sourceName="C" pertinence="1" idRuleParent="" default="true">
        <Description><![CDATA[C=>ball]]></Description>
        <Destinys>
          <Destiny name="ball" var="object"/>
        </Destinys>
      </Rule>
    </Rules>
  </Grammar>
</Compiler>
```

Na figura 5.22 observa-se a configuração definido o algoritmo CYK, utilizado para o processamento de gramáticas probabilísticas, para isso a gramática definida está na forma normal de CHOMSKY.

Na figura 5.23 é possível observar o processamento da cadeia “geeeet ball dog” realizando o processamento como gramática probabilística.

Figura 5.23 – Processamento cadeia “geeeet ball dog”



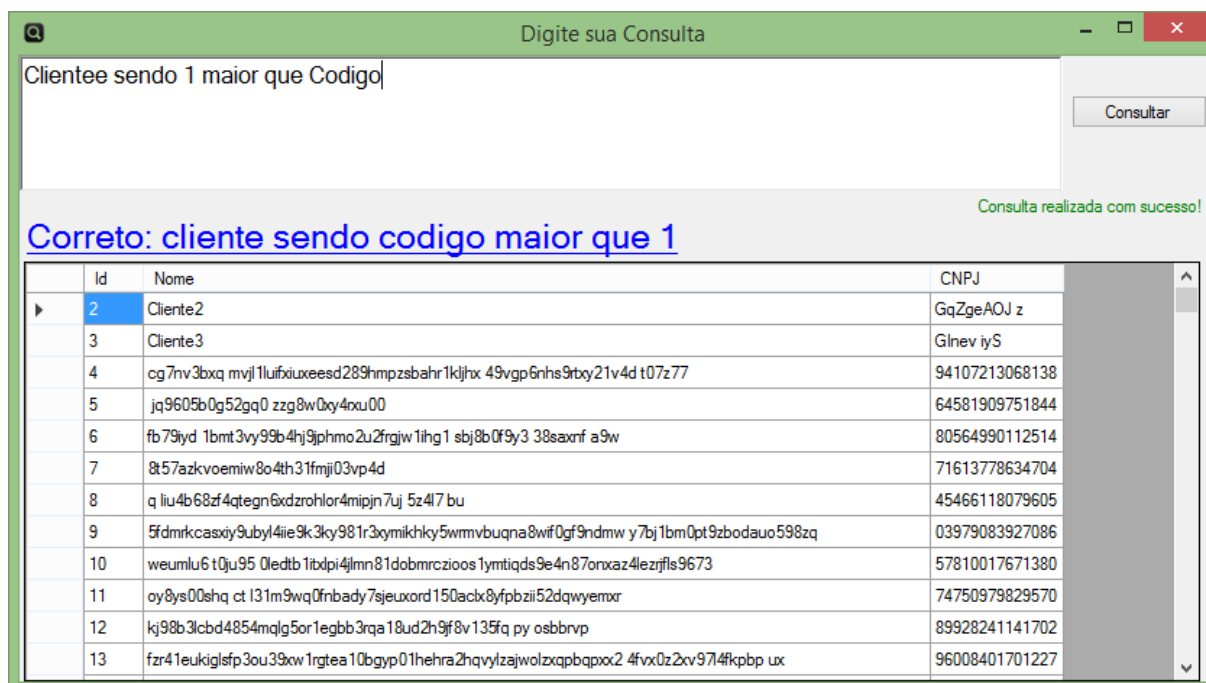
Embora os resultados sejam parecidos entre o processamento probabilístico e com conceitos *fuzzy*, é importante destacar a dificuldade adicional que a definição de uma gramática no formato de CHOMSKY pode trazer para o usuário final da ferramenta.

6 FERRAMENTA DE CONSULTA EM BANCO DE DADOS FACILITADA

Para o teste completo e prova de conceito do *framework*, desenvolveu-se um aplicativo para consulta em banco de dados com uma linguagem de consulta facilitada para usuários com pouco conhecimento em programação.

Na figura 6.1 é demonstrado a tela inicial do programa com uma consulta simples realizada.

Figura 6.1 – Tela principal da aplicação de Consulta



Utilizando o *framework* desenvolvido converteu-se o código de origem “Clientee sendo 1 maior que Codigo” para o código de destino em sql “SELECT * FROM CLIENTE WHERE id > 1”.

Ainda na figura 6.19 é evidente o erro léxico e sintático. A palavra “Clientee” na verdade corresponde a palavra “Cliente” e a ordem da condição “1 maior que código” corresponde a “codigo maior que 1”.

Em azul está destacado a correção sugerida e executada, trazendo os itens listados na tabela.

7 CONCLUSÃO

Este trabalho propôs um estudo sobre as diferentes etapas de um compilador, os conceitos *fuzzy* que podem ser aplicados, o desenvolvimento de uma ferramenta que auxilie na definição e construção de compiladores que utilizem desse conceito e, além disso, a implementação de uma aplicação que utilize de todos os conceitos abordados.

Através da proposta estabelecida pode-se realizar o entendimento da ferramenta desenvolvida.

Com o estudo realizado pode-se perceber o que o conceito de lógica nebulosa pode proporcionar a um compilador convencional. Tolerância e recuperação de erros, sintáticos e léxicos, torna o compilador mais poderoso e as inúmeras aplicações de um compilador pode-se estender, ainda mais, com os conceitos apresentados e disponibilizados através do *framework* desenvolvido.

7.1 Perspectivas Futuras

Em futuras versões do *framework* pode-se explorar mais a fundo os conceitos e algoritmos de gramática probabilística e o desenvolvimento de aplicações mais complexas, como: uma linguagem de programação juntamente com um ambiente de desenvolvimento que tolere e sugira correções nas falhas de programadores iniciantes.

Outras melhorias que podem ser mencionadas se relacionam ao desenvolvimento de outros tipos de analisadores sintáticos e léxicos, além de, novas implementações de norma e conorma para que facilite o usuário final da ferramenta, isolando-o das responsabilidades de implementação e atribuindo-o apenas tarefas de configurações de novos comportamentos no processo de compilação.

REFERÊNCIAS

- ABAR, C. o Conceito “FUZZY”. Disponível em: <http://www.pucsp.br/~logica/Fuzzy.htm>. Acesso em: 13 outubro 2015.
- AHO, A. V.; Lam, M. S.; Sethi R.; Ullman, J. D. **Compiladores, Princípios, técnicas e ferramentas**. 1. ed. São Paulo: Cengage Learning, 2004. 569 p.
- APPEL, A. W.; GINSBURG M. **Modern Compiler Implementation in C**. 1.ed. Cambridge :The Edinburgh Building, 1998. 190 p.
- ARRUDA, D. M.; ABUD, G. M. D.; PONTES, F. A.; PONTES, R. M.; OLIVEIRA, B. B. F. de. Análise comparativa de ferramentas computacionais para modelagem de lógica fuzzy. In SEGet 2013, Rezende, RJ. **Anais.**: Rezende, RJ, 2013;
- CARVALHO, P.; OLIVEIRA, N.; HENRIQUES, P. R. Unfuzzifying Fuzzy Parsing. **3rd Symposium on Languages, Applications and Technologies**, Dagstuhl, Germany, v. 2014 p. 101--108, 2014.
- CAUSSEY, R. L. **Logic, Sets, and Recursion**. 2 ed. Boston: Jones and Bartlett Pub, 1994. 512 p.
- DAVIS, E. CYK-PARSE algorithm with tree recovery. Disponível em: <http://cs.nyu.edu/faculty/davise/ai/CykParse.html>. Acesso em: 16 novembro 2015
- GESSER, C. E. **GALS - Gerador de analisadores léxicos e sintáticos**. 2003. 150 f. Monografia (Bacharel Ciência da Computação) – Universidade Federal de Santa Catarina, Florianópolis, 2003.
- HAHN, K. **Investigation of a fuzzy grammar for automated visual inspection**. 1989. 283 f. Dissertação (Doctor of Philosophy) – Texas Tech University, Texas, 1989.
- KOPPLER, R. A Systematic Approach to Fuzzy Parsing. **Software - Practice and Experience** p. 637-649, Jan. 1999.
- LOPES, I. L.; PINHEIRO, C. A. M.; SANTOS F. A. O. **Inteligência Artificial**. 1 ed. Rio de Janeiro: Elsevier, 2014. 173 p.
- LOUDEN, K. C. **Compiladores, princípios e práticas**. 2. ed. São Paulo: Pearson Education do Brasil, 2008. 633 p.
- MACIEL, A. **Aplicação de autômatos finitos nebulosos no reconhecimento aproximado de cadeias**. 2006. 63 f. Dissertação (Mestrado em Sistemas Digitais) – Escola Politécnica da Universidade de São Paulo, São Paulo, 2006.

OLIVEIRA, W. R. J.; FERRAMOLA L. F.
<http://www.cin.ufpe.br/~if114/Monografias/Automatos%20Probabilisticos/gramatica.htm>. Acesso em: 13 novembro 2015.

REZENDE, S. O.; **Sistemas Inteligentes**. 1. ed. São Paulo: Editora Manole Ltda, 2006. 525 p.

RICARTE, I. **Introdução a Compilação**. 1.ed. Rio de Janeiro: Elsevier, 2008. 258 p.

RIGO, S. Análise Léxica. Disponível em:
<http://www.ic.unicamp.br/~sandro/cursos/mc910/slides/cap2-lex.pdf>. Acesso em: 05 setembro 2015.

SAKATA, T. C. Tópicos em Computação - Lista de Exercícios 2 – Linguagem Livre de Contexto. Disponível em: <http://www.li.facens.br/~tiemi/Tc1/lista2.pdf>. Acesso em: 10 setembro 2015.

TSURUOKA Y.; TSUJII, J. Iterative CKY parsing for Probabilistic Context-Free Grammars. Disponível em:
<http://www.staff.icar.cnr.it/ruffolo/progetti/projects/09.Parsing%20CYK/Iterative%20CKY%20parsing%20for%20Probabilistic%20Context-Free%20Grammars--ijcnlp04.pdf>. Acesso em: 08 novembro 2015.

ZADROZNY, B. Processamento Estatístico da linguagem natural. Disponível em:
http://www2.ic.uff.br/~bianca/peln/index_arquivos/Aula20-PELN.pdf. Acesso em: 08 novembro 2015.