UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO

(Bacharelado)

DESENVOLVIMENTO DE UM PROTÓTIPO DE UM GERADOR DE ANALISADOR LÉXICO

TRABALHO DE CONCLUSÃO DE CURSO SUBMETIDO À UNIVERSIDADE REGIONAL DE BLUMENAU PARA A OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA COM NOME EQUIVALENTE NO CURSO DE CIÊNCIAS DA COMPUTAÇÃO — BACHARELADO

MICHEL NOGUEIRA REBELO

BLUMENAU, NOVEMBRO/2002

2002/2-48

DESENVOLVIMENTO DE UM PROTÓTIPO DE UM GERADOR DE ANALISADOR LÉXICO

MICHEL NOGUEIRA REBELO

ESTE TRABALHO DE CONCLUSÃO DE CURSO FOI JULGADO ADEQUADO PARA OBTENÇÃO DOS CRÉDITOS NA DISCIPLINA DE TRABALHO DE CONCLUSÃO DE CURSO OBRIGATÓRIA PARA OBTENÇÃO DO TÍTULO DE:

BACHAREL EM CIÊNCIAS DA COMPUTAÇÃO

_____

Prof. Joyce Martins — Orientadora na FURB

_____

Prof. José Roque Voltolini da Silva — Coordenador do TCC

BANCA EXAMINADORA

_____

Prof. Joyce Martins

_____

Prof. Jomi Fred Hübner

_____

Prof. José Roque Voltolini da Silva

DEDICATÓRIA

Dedico a Deus, em forma de agradecimento, por estar aqui neste momento com saúde, alegria e sabedoria, por tudo que tenho e que sou, por ser inteligente, por amar e ser amado, por acreditar e ter paciência, por sonhar e ter fé, por ver crescer uma semente, e tudo que estar por vir e for de sua permissão.

AGRADECIMENTOS

A Deus, à professora e orientadora deste trabalho, Joyce Martins, por sua dedicação, apoio, paciência e tolerância, aos meus amigos Otávio Henrique Pereira, Fabrício Bento e Sidnei Teixeira, pela ajuda na parte de implementação e documentação deste, às pessoas que compreenderam a relevância deste trabalho pra mim, respeitando-me e ajudando-me a tomar decisões, e a todos aqueles que contribuíram de alguma forma para a realização deste trabalho.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

AF – Autômato Finito

AFD – Autômato Finito Determinístico

AFND – Autômato Finito Não Determinístico

AFε – Autômato Finito com Movimentos Vazios

ASCII - *American Standard Code for Information Interchange*

BNF – *Backus-Naur Form*

DT – Diagrama de Transição

ER – Expressão Regular

GLC – Gramática Livre de Contexto

GT – Grafo de Transição

HTML – *HyperText Markup Language*

IFPS3 – *Innerfuse Pascal Script III*

LL – Reconhecimento da esquerda para direita (*Left to right)* com derivação mais à esquerda (*Leftmost derivations*).

RAD – *Rapid Application Development*

VCL – *Visual Component Library*

RESUMO

O presente trabalho descreve o desenvolvimento de um protótipo de um gerador de analisador léxico, uma ferramenta que tem como entrada uma especificação, formada por uma lista de palavras reservadas, uma lista de símbolos especiais e definições regulares, de acordo com a BNF de uma meta-linguagem especificada, e produz como saída uma classe que implementa o analisador léxico correspondente. Para a transformação das definições regulares em autômatos finitos é utilizado um algoritmo desenvolvido por Silva (2000). A classe gerada poderá ser integrada facilmente as partes de um compilador por se apresentar em um módulo distinto.

ABSTRACT

The present work describes the development of a prototype of a lexical analyzer generator, a tool that has as entrance a specification, formed by a list of reserved words, a list of special symbols and regular definitions, according to the BNF of a specified meta-language, and it produces as an output a class that implements the corresponding lexical analyzer. For the transformation of the regular definitions in finite automata an algorithm developed by Silva (2000) is used. The generated class will easily be able to be integrated to the parts of a compiler by presenting itself in a distinct module.

1 INTRODUÇÃO

Partindo do princípio de que uma linguagem de programação serve como o meio de comunicação entre o indivíduo que deseja resolver um determinado problema e o computador escolhido para ajudá-lo na solução, pode-se dizer que a linguagem de programação deve fazer a ligação entre o pensamento humano e a precisão requerida para o processamento da máquina.

O desenvolvimento de um programa, segundo Price (2001), torna-se mais fácil se a linguagem de programação em uso estiver mais próxima do problema a ser resolvido, ou seja, se a linguagem inclui construções, através de abstrações, que refletem a terminologia e/ou os elementos usados na descrição do problema. Tais linguagens são denominadas linguagens de alto nível. Já os computadores aceitam e interpretam somente sua própria linguagem de máquina, denominada de baixo nível, a qual consiste tipicamente de seqüências de zeros e uns. As de alto nível são consideradas mais próximas às linguagens naturais ou ao domínio da aplicação em questão. Para que um computador "entenda" os comandos de uma linguagem de alto nível, escritos pelo programador, estes comandos têm que ser traduzidos para uma linguagem de baixo nível, ou seja, linguagem de máquina. Segundo Martins (2002), essa tradução é feita por programas denominados processadores de linguagem, sendo que os principais tipos de processadores são os interpretadores e os tradutores.

Conforme Martins (2002), um interpretador é um programa que aceita como entrada um programa escrito em uma linguagem denominada linguagem fonte, executando-o sem gerar um programa em linguagem de máquina. Já um tradutor é um programa que aceita como entrada um programa escrito em uma linguagem fonte de alto ou de baixo nível, e transforma em um programa escrito em uma outra linguagem, denominada linguagem objeto, que também pode ser uma linguagem de alto ou de baixo nível. Os tradutores podem ser classificados em: montadores, que traduzem linguagens de baixo nível; compiladores, que traduzem linguagens de alto nível; pré-processadores, que traduzem uma linguagem de alto nível em uma outra linguagem de alto nível; *cross-compilers*, que geram código para outra máquina diferente da utilizada na compilação. É importante salientar que para traduzir um programa fonte em um programa objeto pode ser necessário o uso de vários processadores de linguagens.

Conforme Aho (1995), à primeira vista, a variedade de tradutores e, mais especificamente, de compiladores pode parecer assustadora, pois existem milhares de linguagens fontes, que vão das linguagens de programação tradicionais às linguagens especializadas das várias áreas de aplicação de computadores. Por sua vez, as linguagens objetos também são igualmente variadas, podendo ser uma outra linguagem de programação ou uma linguagem de máquina, para qualquer hardware entre um microprocessador e um supercomputador. Apesar de complexas, as tarefas básicas que qualquer compilador precisa realizar são essencialmente as mesmas, de forma que é possível construir compiladores para uma ampla variedade de linguagens fontes e máquinas objetos, usando as mesmas técnicas básicas.

Logo após a escrita dos primeiros compiladores, segundo Aho (1995), surgiram ferramentas para auxiliar esse processo, as quais são freqüentemente referidas como compiladores de compiladores, geradores de compiladores e sistemas de escrita de tradutores.

Este trabalho apresenta o desenvolvimento de outro compilador de compilador, mais especificamente, um protótipo de um gerador de analisador léxico, uma ferramenta de auxílio, que a partir de uma especificação baseada em expressões regulares e em uma lista de palavras reservadas e símbolos especiais, gera automaticamente uma classe que implementa toda a análise léxica de um compilador, na linguagem Object Pascal. A classe gerada reconhece em uma seqüência de cadeias de caracteres, os símbolos léxicos que formam as unidades básicas de um código fonte, também conhecidos por *tokens.*

Para que a classe gerada reconheça os *tokens*, será utilizado um algoritmo descrito por Silva (2000) e implementado por Glatz (2000), que transforma expressões regulares em autômatos finitos. Segundo Glatz (2000), o algoritmo proposto apresenta-se como uma nova solução para minimizar problemas de desempenho e alto consumo de recursos computacionais na transformação de expressões regulares em autômatos finitos. Pode-se citar como relevância deste trabalho que nenhuma das ferramentas geradoras de compiladores mencionadas utilizam o algoritmo descrito por Silva (2000).

O protótipo desenvolvido poderá ser usado pelos acadêmicos da disciplina de Compiladores do curso de Ciências da Computação da FURB, na construção do analisador léxico do compilador para uma linguagem de programação especificada como trabalho da

referida disciplina, o que vem acontecendo normalmente. Objetiva-se abstrair a complexidade dos algoritmos usados, produzindo um analisador léxico que possa facilmente ser integrado às partes restantes do compilador.

1.1 OBJETIVO DO TRABALHO

O objetivo o principal deste trabalho é o desenvolvimento de um protótipo de um gerador de analisador léxico na linguagem Object Pascal (ambiente Delphi), tendo como objetivos mais específicos:

a) definir uma meta-linguagem para as expressões regulares;
b) possibilitar a entrada das expressões regulares, das palavras reservadas e dos símbolos especiais;
c) transformar as expressões regulares em autômatos finitos utilizando o algoritmo descrito por Silva (2000) e implementado por Glatz (2000);
d) gerar uma única classe de objetos que implementa tais autômatos finitos;
e) desenvolver um ambiente de testes e demonstrações.

1.2 ESTRUTURA DO TRABALHO

Este trabalho está dividido em quatro capítulos. O primeiro capítulo é a introdução que contém a origem do trabalho, justificativas e objetivos.

No segundo capítulo são apresentados os tópicos básicos e essenciais para contextualização do presente trabalho de uma maneira direta, sem entrar muito em detalhes nos tópicos de menos relevância, buscando facilitar o entendimento. Neste capítulo é descrito o que é um compilador, quais e como são suas fases e o que são ferramentas para construção de compiladores. Também são apresentados os conceitos de definições regulares e autômatos finitos, bem como algoritmos que transformam expressões regulares em autômatos finitos.

O terceiro capítulo trata dos requisitos, da modelagem e da implementação do protótipo, apresentando desde técnicas e ferramentas utilizadas, como o *Coco/R for Delphi* e o algoritmo descrito por Silva (2000), até operacionalidades da implementação, resultados e discussões finais.

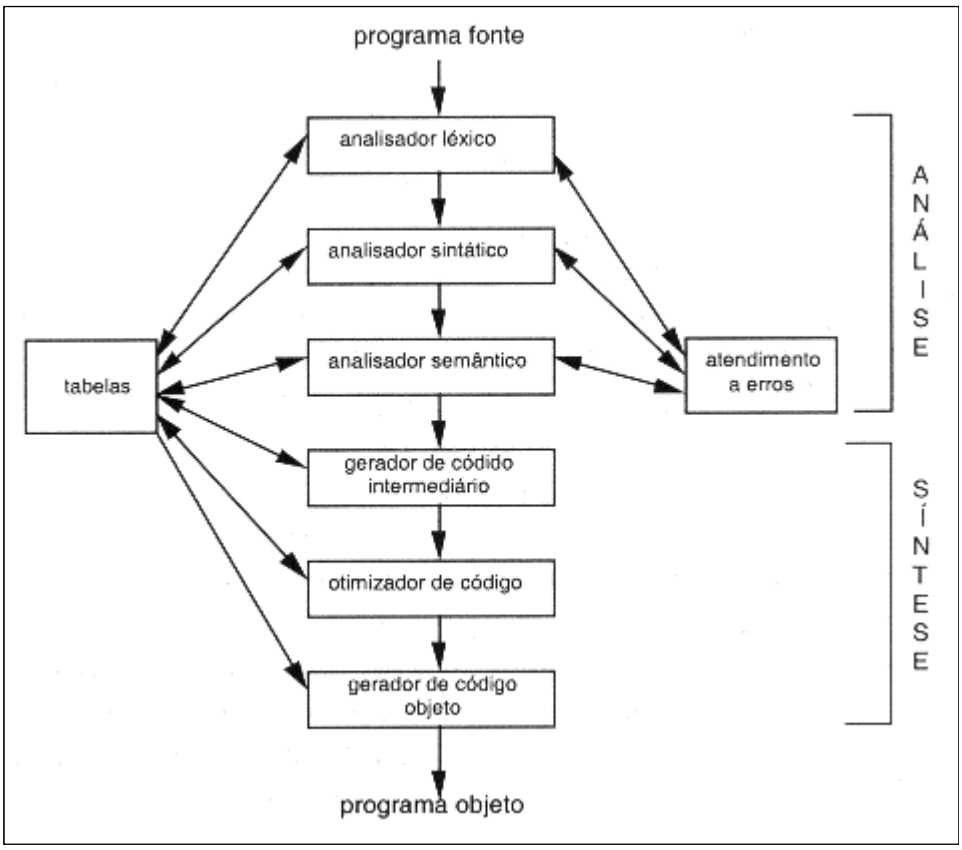
No quarto e último capítulo são relatadas as considerações finais sobre o trabalho e algumas extensões para dar continuidade ao mesmo.

2 FUNDAMENTAÇÃO TEÓRICA

2.1 COMPILADORES

Conforme Grune (2001), de uma forma bem abrangente, um compilador é um programa que tem como entrada um texto escrito em uma determinada linguagem e produz como saída outro texto, escrito em uma outra linguagem, sem perder o significado do texto original. Se o texto estiver em linguagem natural, esse processo é chamado de tradução, por isso, os compiladores também são denominados tradutores. O principal motivo para traduzir um texto é o fato de haver um hardware que execute tal tradução, ou mais precisamente fazer com que o hardware execute as ações descritas pela semântica do programa fonte. De uma maneira geral, um compilador pode ser dividido em duas partes distintas, uma denominada de análise, que faz a consistência do texto de entrada, e outra denominada síntese, que gera o texto de saída. Um exemplo de estrutura de um compilador pode ser visto na fig. 2.1.

FIGURA 2.1 – ESTRUTURA DE UM COMPILADOR



Fonte: Price (2001)

2.1.1 ANÁLISE

Conforme visto na fig. 2.1, a análise é constituída de três fases denominadas análises léxica, sintática e semântica, que comunicam-se umas com as outras para executar a consistência do código fonte ou do texto de entrada. Estas fases podem ser implementadas em módulos individuais, ficando bem mais fácil a visualização de cada uma, ou estar implementadas em um único módulo, gerando desta forma, uma dificuldade na distinção entre elas.

O objetivo principal da primeira fase, a análise léxica, é identificar as seqüências de caracteres que constituem as unidades léxicas, denominadas *tokens*. O analisador léxico lê o programa fonte caracter a caracter, verificando se os caracteres lidos pertencem ao alfabeto da linguagem, identificando os *tokens* e desprezando comentários e espaços em branco desnecessários. Os *tokens* são classificados em categorias, que podem ser basicamente palavras reservadas, identificadores, símbolos especiais e constantes. Além da identificação de *tokens*, o analisador léxico, em geral, inicia a construção da tabela de símbolos e gera mensagens de erro quando identifica *tokens* não aceitos pela linguagem em questão. A saída do analisador léxico é uma lista de *tokens* enumerados e com uma descrição, que é passada para a próxima fase. Muitas vezes, o analisador léxico é implementado como uma subrotina que funciona sob o comando do analisador sintático. Para a construção do analisador léxico, poderão ser utilizados em sua especificação os seguintes formalismos: expressões regulares, definições regulares e autômatos finitos, que serão detalhados na seção 2.3.

A análise sintática tem como objetivo consistir a estrutura sintática do texto de entrada, ou seja, de acordo com a estrutura gramatical especificada, verificar *token* a *token*, se os mesmos estão na seqüência correta. A análise sintática verifica a seqüência dos *tokens*, segundo Price (2001), “através de uma varredura ou *parsing* da representação interna (cadeia de *tokens*) do programa fonte”. Para representar a estrutura sintática do texto fonte é, explicita ou implicitamente, produzida uma estrutura de árvore, denominada árvore sintática. Segundo Price (2001), “outra função dos reconhecedores sintáticos é a detecção de erros de sintaxe, identificando clara e objetivamente a posição e o tipo de erro ocorrido”. Na especificação da sintaxe de uma linguagem de programação pode ser utilizado como formalismo gramática livre de contexto (GLC), pois tal gramática, segundo Price (2001), “permite descrever a maioria das linguagens de programação usadas atualmente”. De acordo com Martins (2002),

uma GLC é uma quádrupla representado por G = (V_N, V_T, P, S), onde V_N é um conjunto de símbolos não terminais; V_T é um conjunto de símbolos terminais; P é um conjunto de regras gramaticais, também chamadas de produções; S é o símbolo inicial da gramática. As produções relacionam símbolos terminais e símbolos não terminais, especificando a seqüência correta dos símbolos terminais, ou seja, dos *tokens* identificados pelo analisador léxico. Cada uma destas regras possui um símbolo não terminal seguido por ⇒ ou por ::=, que separa o símbolo do restante da produção. No restante da produção poderá existir uma seqüência de símbolos terminais ou não terminais, inclusive o símbolo vazio, representado por ε. Para a representação de uma GLC é utilizada, na maioria das vezes, uma notação conhecida por BNF (*Backus-Naur Form*), que segundo Price (2001), popularizou o uso das GLCs. Um exemplo de uma GLC representada com a notação BNF pode ser visto no capítulo 3.

Na análise semântica, segundo Price (2001), é verificado se as estruturas do programa irão fazer sentido durante a execução. Assim, por exemplo, a um identificador declarado como de um determinado tipo só podem ser atribuídos valores deste mesmo tipo; operandos e operadores devem ser compatíveis em expressões etc. A entrada para o analisador semântico é a árvore sintática produzida pela análise sintática, sendo que, muitas vezes, o analisador semântico opera juntamente com o analisador sintático. Segundo Martins (2002), para a especificação da semântica de uma linguagem de programação, o formalismo é mais complexo e de difícil aprendizagem, podendo ser realizado “através de métodos formais tais como semântica operacional, semântica de ações, semântica axiomática ou semântica denotacional; ou de maneira semi-formal através de ações semânticas embutidas na gramática da linguagem em questão”.

2.1.2 SÍNTESE

Segundo Price (2001), na síntese é construído o código objeto a partir da representação intermediária. A síntese divide-se em gerador de código intermediário, otimizador de código e gerador de código objeto. O gerador de código intermediário utiliza a representação interna produzida pela análise sintática para gerar um outro código, chamado código intermediário. O otimizador, como o próprio nome diz, tem por objetivo otimizar o código intermediário a fim de obter um código objeto final com maior velocidade de execução e menor espaço em memória. Por fim, o gerador de código objeto, segundo Price

(2001), “é a fase mais difícil, pois requer uma seleção cuidadosa das instruções e dos registradores da máquina alvo”. Pode-se citar como objetivos, além da produção de código objeto eficiente, a reserva de memória para constantes e variáveis e a seleção de registradores.

2.2 FERRAMENTAS PARA CONSTRUÇÃO DE COMPILADORES

Conforme Aho (1995), ferramentas para construção de compiladores são ferramentas que auxiliam na construção das rotinas que constituem um compilador e são conhecidas como compiladores de compiladores, geradores de compiladores ou sistemas de escrita de tradutores. São ferramentas amplamente orientadas em torno de um modelo particular de linguagem e mais adequadas para gerar compiladores de linguagens similares ao modelo. Conforme Price (2001), tais ferramentas classificam-se em três grupos:

a) **geradores de analisadores léxicos:** segundo Aho (1995), pode-se assumir que os analisadores léxicos para todas as linguagens são iguais, diferenciando apenas o conjunto de palavras reservadas e símbolos especiais, que deverá ser fornecido pelo usuário. Muitas destas ferramentas são criadas a partir de notações baseadas em expressões regulares, que servem para especificar os padrões dos *tokens*, onde um algoritmo é utilizado para compilar tais expressões regulares em programas reconhecedores de *tokens*, que validam se as cadeias de entrada fazem parte ou não de uma determinada linguagem;

b) **geradores de analisadores sintáticos:** segundo Price (2001), normalmente, é a partir de uma gramática livre de contexto, que um gerador de analisador sintático produz um reconhecedor sintático. Geradores de analisadores gramaticais, como são chamados por Aho (1995), usam algoritmos de análise gramatical que são muito complexos para serem realizados a mão;

c) **geradores de geradores de código:** tais ferramentas, segundo Price (2001), “recebem como entrada regras que definem a tradução de cada operação da linguagem intermediária para a linguagem de máquina”, onde tais regras devem ser detalhadas de maneira que possibilitem manipular diferentes métodos de acesso a dados. Pode-se citar como exemplo a alocação de uma variável que poderá ser em memória, em registradores ou em uma pilha da máquina. Segundo Price (2001), “em geral, instruções intermediárias são mapeadas para esqueletos que representam

seqüências de instruções de máquina".

Dentre as muitas ferramentas geradoras de compiladores existentes, pode-se citar: *LISA* (Zumer, 2002), *JavaCC* (WebGain, 2002) e *Coco/R* (Mössenböck, 2002).

Conforme Zumer (2002), *LISA* é uma ferramenta para gerar um compilador para uma linguagem utilizando métodos formais já conhecidos, quais sejam expressões regulares, notação BNF (*Backus-Naur Form*) e gramática de atributos. Possui um ambiente amigável que permite escrever a descrição formal de uma linguagem de uma maneira rápida e simples. *LISA* gera os analisadores léxico, sintático e semântico em C++.

Outra ferramenta bastante conhecida é o *JavaCC*. Conforme WebGain (2002), com *JavaCC* pode ser gerado desde uma simples linguagem para pequenos problemas, até compiladores complexos para linguagens como Java e C++, ou ainda ferramentas que analisam gramaticalmente o código fonte Java e executam automaticamente tarefas de análise ou de transformação. *JavaCC* também gera analisadores léxicos, analisadores sintáticos recursivos descendentes LL(1) e programas gerenciadores de gramáticas de atributos em Java. *JavaCC* inclui duas ferramentas adicionais: JJDoc, que gera documentação em HTML para uma gramática de forma semelhante ao JavaDoc, e JJTree, que gera ações que constróem uma estrutura de árvore automaticamente ao analisar gramaticalmente um programa. Há numerosas outras características, inclusive capacidades de depuração, gerenciador de erros, entre outras.

Já *Coco/R* é uma ferramenta que, conforme Mössenböck (2002), gera compiladores em Oberon, Modula-2, Pascal, Delphi, C/C++ e Java para diferentes plataformas como MS-DOS, Atari, UNIX, Linux e 386BSD. *Coco/R* é uma ferramenta que combina a funcionalidade das ferramentas Lex e Yacc em um compilador de compilador, que gera analisadores léxicos, analisadores sintáticos recursivos descendentes LL(1) e, em algumas versões, um programa gerenciador de gramáticas de atributos. Uma das versões Delphi do *Coco/R* é uma ferramenta chamada *Congencee* que, conforme CocolSoft (2001), permite construir analisadores léxicos, analisadores sintáticos, compiladores, interpretadores, processadores de linguagens naturais, *shell* de sistema especialistas, entre outras aplicações. No entanto, *Congencee* não é uma ferramenta *freeware*. Outra versão Delphi do *Coco/R* é *Coco/R for Delphi* que, conforme

Reith (2002), é um compilador de compilador *freeware*, que aceita como entrada uma gramática livre de contexto e gera um analisador sintático recursivo descendente LL(1).

2.3 ANALISADOR LÉXICO

Como relatado anteriormente, a função do analisador léxico é identificar os *tokens* contidos no texto fonte. Para realizar tal tarefa, um analisador léxico utiliza autômatos finitos que interpretam as expressões ou as definições regulares especificadas pelo usuário. Assim, para construir um analisador léxico para uma linguagem de programação, basicamente é necessário: especificar as expressões regulares, definir autômatos finitos correspondentes e implementar os autômatos em alguma linguagem de programação.

2.3.1 ESPECIFICAÇÃO DOS *TOKENS*: EXPRESSÕES E DEFINIÇÕES REGULARES

Segundo Aho (1995), as expressões regulares (ER) são uma notação importante para especificar padrões, sendo que cada padrão corresponde a um conjunto de cadeias e uma cadeia é uma seqüência finita de símbolos retirados de um alfabeto. As expressões regulares podem ser definidas pelas seguintes regras:

a) $\varnothing$ é uma ER que denota uma linguagem vazia;
b) ε é uma ER que denota uma linguagem contendo o símbolo vazio;
c) Se x é um símbolo de um alfabeto, x é uma ER que denota uma linguagem contendo o símbolo x como palavra;
d) Se r e s são ER denotando as linguagens R e S, então:
   - (r | s) é uma ER que denota uma linguagem $R \cup S$;
   - (rs) é uma ER que denota a linguagem RS, onde $r \in R$ e $s \in S$;
   - (r*) é uma ER que denota a linguagem R*.

Quando as expressões regulares são identificadas por nomes e esses nomes são utilizados como símbolos em outras expressões, tem-se o conceito de definições regulares. No exemplo do quadro 2.1, tem-se a especificação de identificadores em uma linguagem de programação qualquer. Pela especificação feita, um *identificador* é um *token* que inicia com uma *letra* e os demais caracteres da seqüência podem ser outras letras, dígitos ou o caracter

underline ("_"), onde *letra* identifica o conjunto de caracteres de "a" a "z", podendo ser letra maiúscula ou minúscula, e *dígito* é o conjunto de caracteres de "0" a "9".

QUADRO 2.1 – DEFINIÇÃO REGULAR PARA IDENTIFICADORES

letra → a | b | .. | z | A | B | .. | Z
dígito → 0 | 1 | .. | 9
identificador → letra (letra | dígito | _)*

2.3.2 RECONHECIMENTO DE *TOKENS*: AUTÔMATOS FINITOS

Um autômato finito (AF) é o tipo mais simples de reconhecedor de linguagens existente, podendo ser utilizado para reconhecer *tokens* de um texto fonte. Segundo Aho (1995), um autômato finito é um modelo natural em torno do qual pode-se construir um analisador léxico. É uma máquina de estados finitos que valida os *tokens* através de uma série de transições de estado. De acordo com Lewis (2000), "não há saída, não há memória e a capacidade de processamento é fixa e finita". Um estado é uma situação particular no processo de reconhecimento de uma cadeia e uma transição de estado é uma alteração de um estado para outro. Desta forma, Martins (2002) diz que um AF tem capacidade de observar apenas um símbolo de cada vez e assume-se que a máquina, ao iniciar sua operação, esteja no estado inicial. A entrada de um AF é uma seqüência de símbolos que são lidos um a um. Segundo Martins (2002), "o autômato muda seu estado apenas em função do estado corrente e do símbolo de entrada", ocorrendo assim, uma transição de estado. Um AF possui estados denominados finais que são utilizados para verificar se a cadeia pertence ou não à linguagem, sendo que se após lidos todos os símbolos da cadeia, o autômato parar em um dos estados finais é porque a cadeia foi reconhecida, caso contrário, a cadeia de entrada é inválida. Se para algum símbolo da seqüência não existir transição, o autômato pára e tal seqüência também não é reconhecida. O processo de um AF é feito através de operações, que podem ser descritas pela função matemática δ (e_{velho}, x) = e_{novo}, chamada função de transição, que, conforme Martins (2002), "determina o estado e_{novo} em função do estado e_{velho} e do símbolo de entrada x". Os autômatos finitos podem ser classificados em determinístico (AFD), não-determinísticos (AFND) e com movimentos vazios (AFε).

2.3.2.1 FORMAS DE REPRESENTAÇÃO DE UM AUTÔMATO FINITO

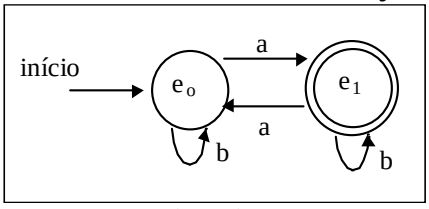
Segundo Martins (2002), um AF é um sistema formal M = (K, $\sum$, δ, e_0, F), onde K é um conjunto finito não-vazio de estados, $\sum$ é o alfabeto de símbolos de entrada, δ é a função de transição, e_0 é o estado inicial ($e_0 \in$ K) e F é o conjunto de estados finais (F $\subseteq$ K), podendo ser representado através de diagramas de transição (DT) ou grafos de transição (GT), como também são conhecidos. Conforme pode ser visto nas fig.2.2, 2.3 e 2.4, um diagrama de transição para um autômato finito é um grafo direcionado e rotulado, onde os nodos são os estados, fisicamente representados por círculos, sendo que o estado inicial é precedido por uma seta com rótulo "início" e os estados finais são representados por círculos duplos. As arestas representam as transições e são rotuladas com os símbolos do alfabeto de entrada. Tais diagramas ou grafos de transição podem representar qualquer tipo de autômato finito.

Outra maneira de representar um AF é com tabelas de transição. Conforme pode ser visto nas tabelas 2.2, 2.3 e 2.4, é uma forma de representação tabular onde, segundo Martins (2002), "as linhas representam os estados (o inicial é indicado por uma seta e os finais por asteriscos), as colunas representram os símbolos de entrada e o conteúdo da posição (e_0, x) será igual a e_1 caso exista δ (e_0, x) = e_1, caso não exista, será indefinida".

2.3.2.2 AUTÔMATOS FINITOS DETERMINÍSTICOS

Segundo Martins (2002), em um AFD, a cada símbolo reconhecido o estado atual pode somente ser alterado para um único outro estado, sem existir indeterminismo, ou seja, sem que haja mais de um estado alternativo. Um exemplo de AFD para a linguagem composta por **a**s e **b**s, com no mínimo um símbolo e cujo número de **a**s é ímpar, é mostrado na fig.2.2 e a tabela de transição correspondente encontra-se na tabela 2.2.

FIGURA 2.2 – DIAGRAMA DE TRANSIÇÃO DE UM AFD



Fonte: Martins (2002)

TABELA 2.1 – TABELA DE TRANSIÇÃO DE UM AFD

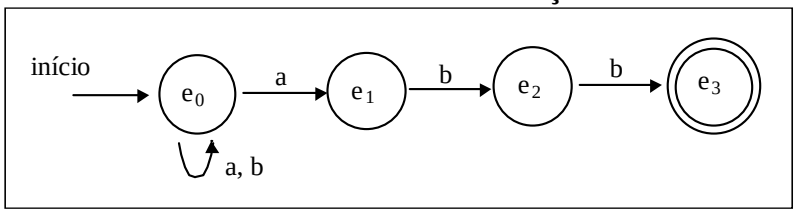
	δ	a	b
→	e_0	e_1	e_0
*	e_1	e_0	e_1

Fonte: Martins (2002)

2.3.2.3 AUTÔMATOS FINITOS NÃO-DETERMINÍSTICOS

Segundo Martins (2002), se em um autômato, para certo símbolo da entrada, existir mais de uma opção para um próximo estado, o autômato deixa de ser determinístico e é classificado como não determinístico ou, mais especificamente, após reconhecer x o próximo estado pode ser um dentre vários estados diferentes. Um exemplo de AFND é mostrado na fig. 2.3 e na tabela 2.3, onde a linguagem reconhecida é definida pela expressão regular (a | b)* a b b.

FIGURA 2.3 – DIAGRAMA DE TRANSIÇÃO DE UM AFND



Fonte: Martins (2002)

TABELA 2.2 – TABELA DE TRANSIÇÃO DE UM AFND

	δ	a	b
→	e_0	e_0, e_1	e_0
	e_1	-	e_2
	e_2	-	e_3
*	e_3	-	-

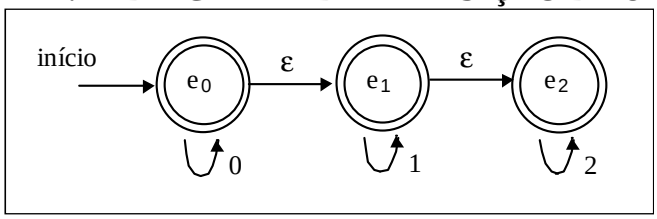
Fonte: Martins (2002)

2.3.2.4 AUTÔMATOS FINITOS COM MOVIMENTOS VAZIOS

Segundo Martins (2002), um autômato com movimento vazios é um AFND que aceita o símbolo vazio (ε) como entrada, ou seja, efetua “transições com entrada vazia”, “interpretadas como um não determinismo interno ao autômato”. Um exemplo pode ser visto na fig. 2.4 para a linguagem definida pela expressão regular (0* 1* 2*). Na tabela 2.4 pode ser

observada a utilização de uma coluna com o símbolo vazio (ε), representando as transições rotuladas com o símbolo vazio (ε) nas arestas entre cada dois estados do diagrama de transição.

FIGURA 2.4 – DIAGRAMA DE TRANSIÇÃO DE UM AFε



Fonte: Martins (2002)

TABELA 2.3 – TABELA DE TRANSIÇÃO DE UM AFε

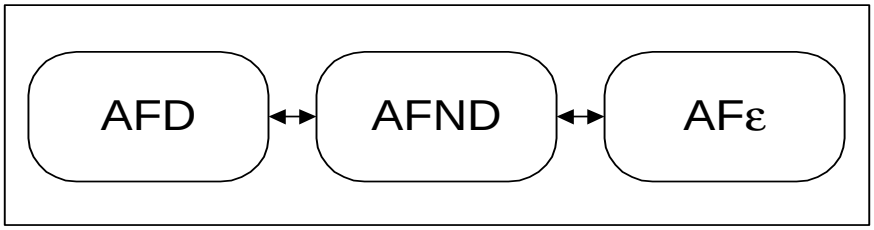
	δ	0	1	2	ε
* →	e_0	e_0	-	-	e_1
*	e_1	-	e_1	-	e_2
*	e_2	-	-	e_2	-

Fonte: Martins (2002)

2.3.2.5 EQUIVALÊNCIA ENTRE AUTÔMATOS FINITOS

De acordo com Glatz (2000), os vários tipos de autômatos finitos permitem a equivalência entre si, conforme mostra a fig. 2.5.

FIGURA 2.5 – EQUIVALÊNCIA ENTRE AUTÔATOS FINITOS



Fonte: Glatz (2000)

Pode-se mostrar que, “a partir de um AFND, é possível construir um AFD que realiza o mesmo processamento” (Glatz, 2000), através de um algoritmo que constrói um AFD equivalente a um AFND, contendo todas as alternativas existentes no AFND. Pode-se mostrar também que um AFND é equivalente a um AFε, construindo-se um AFND que realiza o mesmo processamento do AFε, produzindo-se uma função de transição sem movimentos

vazios, onde o conjunto de novos estados de cada transição não vazia é estendido com todos os estados possíveis de serem atingidos por transições vazias (ε) (Menezes, 1998).

2.3.3 ALGORITMOS PARA IMPLEMENTAÇÃO DE AUTÔMATOS FINITOS

Segundo Glatz (2000), “para toda expressão regular R sobre o ∑ existe um autômato finito M sobre o ∑ tal que o conjunto de todas as palavras reconhecidas por M é igual ao conjunto de todas as palavras reconhecidas por R”. Deste modo, pode-se construir algoritmos genéricos que transformam qualquer ER em um AFD, baseando-se no Teorema de Kleene que “define que qualquer conjunto reconhecido por uma máquina de estados finitos é regular e qualquer conjunto regular pode ser reconhecido por uma máquina de estados finitos” (Gersting, 1995). Em Glatz (2000), foram citados três algoritmos que transformam ER em AFD, descritos por Manna (1974), Hopcroft (1979) e Silva (2000) respectivamente; e foram implementados e comparados os algoritmos descritos por Hopcroft (1979) e por Silva (2000), chegando a conclusão que o algoritmo descrito por Silva (2000) é mais eficiente em vários aspectos, como na utilização de memória, tempo de processamento e otimização. Assim sendo, no protótipo desenvolvido será utilizado para a implementação dos autômatos finitos, o algoritmo descrito por Silva (2000), detalhado no capítulo seguinte.

3 DESENVOLVIMENTO DO TRABALHO

As definições descritas no capítulo anterior são de fundamental importância para o desenvolvimento deste trabalho, pois é de extrema necessidade o conhecimento sobre compiladores, ferramentas geradoras de compiladores e, principalmente, sobre expressões regulares e autômatos finitos, que são os formalismos utilizados na construção de um analisador léxico.

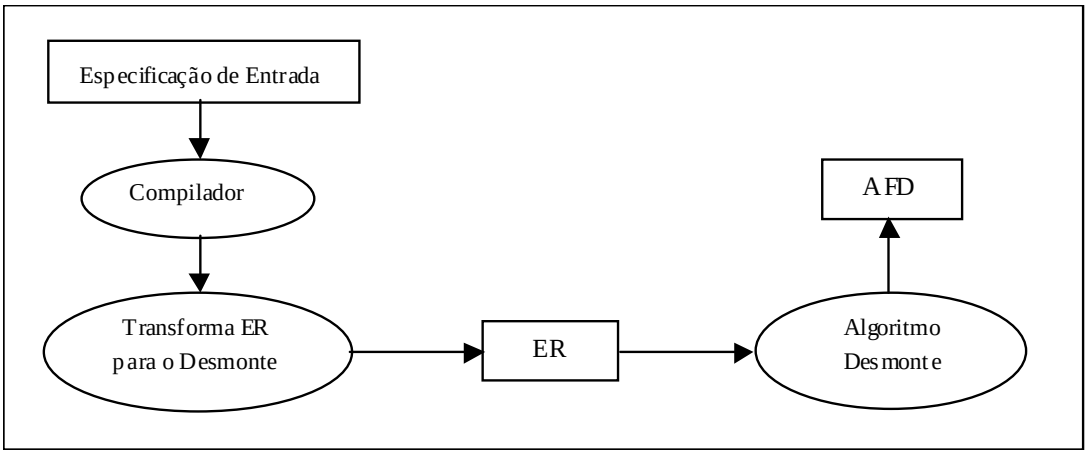
Procurou-se desenvolver a ferramenta proposta utilizando o desenvolvimento rápido de aplicações (RAD) que, segundo Thiry (2001), não é exatamente um modelo mas baseia-se no fato de que “um modelo de ciclo de vida formal é ineficiente e muitas documentações são perda de tempo e dificultam a comunicação com o cliente”. Thiry (2001) diz que “não existe um modelo de ciclo de vida bem definido”, e sim seqüências de integrações evolucionárias, associadas a um período de tempo. Assim, pode-se citar como passos no RAD: analisar os requisitos; desenvolver um projeto inicial; desenvolver dentro de um determinado tempo uma versão da aplicação; entregar a versão para que o cliente possa testar; receber o *feedback* e planejar uma versão para responder a este *feedback* caso a versão apresente problemas. Os quatro últimos passos devem ser repetidos até que a aplicação esteja pronta.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

O protótipo desenvolvido terá como entrada a especificação das unidades básicas da linguagem, isto é, a definição dos *tokens*, de acordo com uma meta-linguagem que encontra-se especificada na seção 3.2.1, e como saída uma classe final definida na seção 3.2.2, que implementa um analisador léxico em Object Pascal (ambiente Delphi).

Para a geração da classe final, é necessário primeiro transformar as expressões regulares informadas na especificação de entrada em um único autômato finito que validará os *tokens* que não forem classificados como palavras reservadas ou símbolos especiais. Para tal transformação será utilizado o algoritmo “Desmonte” descrito por Silva (2000), conforme é mostrado na fig. 3.1.

FIGURA 3.1 – TRANSFORMAÇÃO DE EXPRESSÕES REGULARES EM AUTÔMATOS FINITOS



Na execução da classe gerada a partir da especificação de entrada, serão mostrados os *tokens* lidos e os erros encontrados, a partir da identificação das palavras reservadas, dos símbolos especiais e das definições regulares. Para isso, as definições regulares devem ser convertidas para ER de acordo com as definições de Glatz (2000), onde caracteres e literais não possuem aspas, não existe a notação de faixa de valores expressos entre colchetes, e nem os operadores ‘+’, ‘?’ e ‘~’. Para fazer a conversão da especificação de entrada em uma especificação válida para o algoritmo de Silva (2000), implementado por Glatz (2000), será feita uma transformação, conforme é mostrado na tabela 3.1.

TABELA 3.1 – TRANSFORMAÇÃO DA ESPECIFICAÇÃO DE ENTRADA EM UMA ENTRADA VÁLIDA PARA A IMPLEMENTAÇÃO DE GLATZ (2000)

OPERADOR OU NOTAÇÃO	EXEMPLO DE ESPECIFICAÇÃO DE ENTRADA	ESPECIFICAÇÃO TRANSFORMADA
literal	“abc”	abc
caracter	‘a’	A
+	a +	(a a*)
?	a ?	(a | ^)
~	~ [‘#’]	(! | “ | $ | .. todos os caracteres da tabela ASCII) conforme descrito na tabela 2.1, menos o caracter ‘#’
[ ]	[‘a’..’z’]	(a | b | c | . . | z)

O teste de utilização da classe final será feito em tempo de execução, sem precisar compilar a classe gerada para utilizá-la. Já para integrar a classe gerada com os demais módulos de um compilador, será necessário adicionar e compilar junto ao projeto outras duas classes, quais sejam *TArrayTokens* e *TTokens*. Como foi utilizado *array* dinâmico na classe *TArrayTokens,* a mesma só poderá ser compilada a partir da versão 4 do Delphi, pois esta estrutura não é encontrada em versões anteriores.

3.2 ESPECIFICAÇÃO

3.2.1 ESPECIFICAÇÃO DA META-LINGUAGEM

A especificação dos *tokens* de uma nova linguagem deve ser feita utilizando a meta-linguagem criada. Deve-se especificar o conjunto das palavras reservadas, possivelmente vazio, o conjunto dos símbolos especiais que também pode ser vazio e o conjunto dos demais *tokens* não específicos como identificadores e constantes numéricas. Uma palavra reservada ou um símbolo especial é um literal e um *token* é definido por um identificador seguido por dois pontos e por uma definição regular.

Assim, as unidades básicas da meta-linguagem são: identificador, literal e caracter, cuja especificação encontra-se no quadro 3.1.

QUADRO 3.1 – DEFINIÇÕES REGULARES DA META-LINGUAGEM

letra → a | .. | z | A | .. | Z
dígito → 0 | 1 | .. | 9
identificador → letra (_ | letra | dígito)*
literal → " ASCII+ "
caracter → ' ASCII '

onde ASCII representa todos os caracteres válidos da tabela ASCII, exceto os seguintes caracteres: ‘&’, ‘(’, ‘)’, ‘*’, ‘^’, ‘|’

Observa-se que letra e dígito só são utilizados na definição de *identificador*, sendo que um identificador inicia com uma letra e os demais caracteres da seqüência podem ser outras letras, dígitos ou o caracter *underline* ("_"). Já um *literal* é qualquer seqüência composta por um ou mais caracteres da tabela ASCII, menos aspas duplas que são utilizadas para delimitar o literal; enquanto um *caracter* é composto por um único caracter da tabela ASCII, exceto aspas simples que são utilizadas para delimitar o caracter. Conforme é mostrado no quadro

3.1, um *literal* ou um *caracter*, além das restrições das aspas duplas e simples respectivamente, não pode possuir os caracteres do conjunto {‘&’, ‘(’, ‘)’, ‘*’, ‘^’, ‘|’} que, conforme dito anteriormente, são utilizados por Glatz (2000) na implementação do algoritmo de Silva (2000). *Identificador*, *literal* e *caracter* são usados na especificação da meta-linguagem como símbolos terminais de uma gramática livre de contexto escrita na notação BNF.

No quadro 3.2 é mostrado um exemplo da especificação de identificadores correspondentes ao exemplo apresentado no quadro 3.1. utilizando a notação definida.

QUADRO 3.2 – DEFINIÇÃO REGULAR PARA IDENTIFICADORES NO PROTÓTIPO

letra → [‘a’ . . ‘z’] | [‘A’ . . ‘Z’]

dígito → [‘0’ . . ‘9’]

identificador → letra (letra | dígito | _)*

A tabela 3.2 mostra a notação dos operadores que será utilizada na especificação das definições regulares do protótipo implementado.

TABELA 3.2 – NOTAÇÃO PARA DEFINIÇÕES REGULARES

OPERADOR	DESCRIÇÃO
*	indica nenhuma, uma ou muitas ocorrências
+	indica uma ou muitas ocorrências
?	indica nenhuma ou uma ocorrência
~	indica uma negação, ou seja, a ocorrência de todos os caracteres da tabela ASCII exceto os informados posteriormente entre colchetes. Por exemplo, ~ [‘**”**’], indica a ocorrência de todos os caracteres da tabela ASCII exceto aspas duplas. Esse não é um operador padrão. No protótipo “todos os caracteres” são representados pelo conjunto ASCII {CHR(33)..CHR(255)} menos os símbolos ‘&’, ‘(’, ‘)’, ‘*’, ‘^’ e ‘|’, respectivamente CHR(38), CHR(40), CHR(41), CHR(42), CHR(94) e CHR(126), que são utilizados por Glatz (2000) na implementação do algoritmo de Silva (2000).
[ ]	indica a ocorrência de um elemento do conjunto de caracteres especificado entre colchetes, podendo existir faixas ou caracteres únicos separados por vírgula. Por exemplo, [‘a’..‘z’, ‘_’, ‘0’..’9’] representa a ocorrência de um elemento do conjunto que pode ser uma letra minúscula, o *underline* ou um dígito.
|	indica que existe mais de uma alternativa, isto é, corresponde ao operador lógico OU
R S	indica a concatenação, ou seja, corresponde ao operador lógico E, onde R e S representam duas definições regulares. Neste caso, a concatenação é feita automaticamente sem a necessidade de existência de um operador.

No quadro 3.3 é mostrado a sintaxe da meta-linguagem, onde os símbolos não terminais estão especificados entre os sinais de maior e menor, as palavras reservadas e os símbolos especiais estão em negrito e o símbolo vazio é representado por ε.

QUADRO 3.3 – BNF DA META-LINGUAGEM

```
<especificação> ::= PALAVRAS RESERVADAS : <caixa> { <conjunto de literais> }
                    SIMBOLOS ESPECIAIS : { <conjunto de literais> }
                    TOKENS : { <conjunto de tokens> }
<caixa> ::= ( CAIXA SENSÍVEL ) | ε
<conjunto de literais> ::= <lista de literais> | ε
<lista de literais> ::= literal | literal , <lista de literais>
<conjunto de tokens>::= <token> | <token> <conjunto de tokens>
<token>::= < identificador : <expressão regular> >
<expressão regular>::= ( <expressão regular> ) <expressão regular'>
                    | <termo> <expressão regular'>
<expressão regular'>::= <operador> <expressão regular'>
                    | | <expressão regular>
                    | <expressão regular>
                    | ε
<termo>::=          [ <faixa> ]
                    | ~ [ <faixa> ]
                    | identificador
                    | literal
                    | caracter
<operador> ::= + | * | ?
<faixa> ::= <sub-faixa> | <sub-faixa>, <faixa>
<sub-faixa> ::= caracter | caracter .. caracter
```

Conforme é mostrado no quadro 3.2, "*CAIXA SENSÍVEL*" é uma opção que pode ser informada na especificação de entrada antes das palavras reservadas, e tem como finalidade informar ao gerador de analisador léxico que é para gerar uma classe que diferencia as letras maiúsculas das minúsculas no reconhecimento de palavras reservadas. Se não for informada tal opção, o analisador léxico não fará essa diferenciação.

A semântica da meta-linguagem é implementada através de ações embutidas na gramática especificada, que definem as seguintes restrições: não podem ser informadas palavras reservadas iguais, levando em consideração a opção "*CAIXA SENSÍVEL*"; não podem ser informados símbolos especiais iguais ou com mais de 2 caracteres; não podem ser informados identificadores de expressões regulares ou expressões regulares iguais; na

definição de uma expressão regular somente podem ser usados identificadores de expressões regulares definidas anteriormente. Outras restrições semânticas dizem respeito às faixas de valores, representadas pela notação de conjunto "[ ]", onde: em uma sub-faixa o primeiro caracter nunca poderá ser maior que o segundo, por exemplo ['z'..'a'] é uma sub-faixa inválida; os dois caracteres de uma sub-faixa nunca podem ser iguais, como em ['a'..'a']; uma faixa não pode conter o mesmo caracter mais de uma vez, como exemplifica ['a'..'z', 'd'], onde o caracter 'd' já pertence à faixa especificada por 'a'..'z'. Conforme descrito anteriormente e mostrado na tabela 3.1, a transformação da especificação de entrada em expressões regulares válidas para o algoritmo de Silva (2000) implementado por Glatz (2000), também é feita através de ações semânticas, que podem ser vistas no Apêndice 1 as quais estão definidas entre parênteses e ponto , como por exemplo "(. TipoLiteral := tPR; .)".

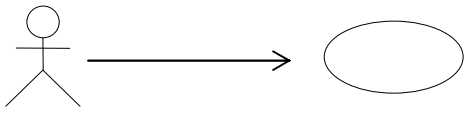
3.2.2 ESPECIFICAÇÃO DO SISTEMA

O protótipo foi modelado utilizando o padrão *Unified Modeling Language* (UML) (Furlan, 1998). Primeiramente, serão apresentados os *use cases* e, em seguida, o diagrama de classe. Na especificação da aplicação foi utilizada a ferramenta *Rational Rose* (Quatrani, 2001).

No quadro 3.4 tem-se o primeiro *use case*[1], que descreve como é feita a especificação dos *tokens* no protótipo.

[1] A notação para *use case* utilizada é uma adaptação daquela apresentada por Thiry (2001).

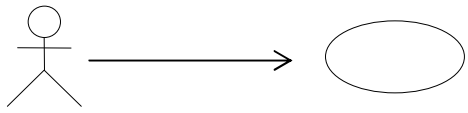
QUADRO 3.4 – *USE CASE:* ESPECIFICAR TOKENS



USE CASE:	Especificar *tokens*
BREVE DESCRIÇÃO:	Em uma janela do protótipo, o usuário irá especificar as unidades básicas da linguagem, ou seja, irá definir os *tokens* utilizando a meta-linguagem definida.
ATOR(ES):	Usuário
PRÉ-CONDIÇÕES:	É necessário que o usuário conheça a meta-linguagem.
FLUXO PRINCIPAL:	1. O usuário entra com a especificação dos *tokens* e seleciona o botão *Compilar*. 2. O protótipo faz toda a análise da especificação de entrada, mostrando os *tokens* lidos e os possíveis erros encontrados. 3. É gerada a classe *TAnalisaLex*, que implementa o analisador léxico correspondente à especificação de entrada.
FLUXOS ALTERNATIVOS E EXCEÇÕES:	No item 2, caso seja detectado um erro léxico, sintático ou semântico, o protótipo emite uma mensagem do tipo: “Foi encontrado erro durante a compilação. A classe não foi gerada!”
PÓS-CONDIÇÕES:	A classe final gerada ou mensagem de erro.
REQUISITOS FUNCIONAIS SATISFEITOS:	O protótipo deve gerar uma classe que implementa o analisador léxico, de tal forma que a classe gerada possa ser adicionada e compilada em qualquer projeto Delphi a partir da versão 4, juntamente com outras duas classes.

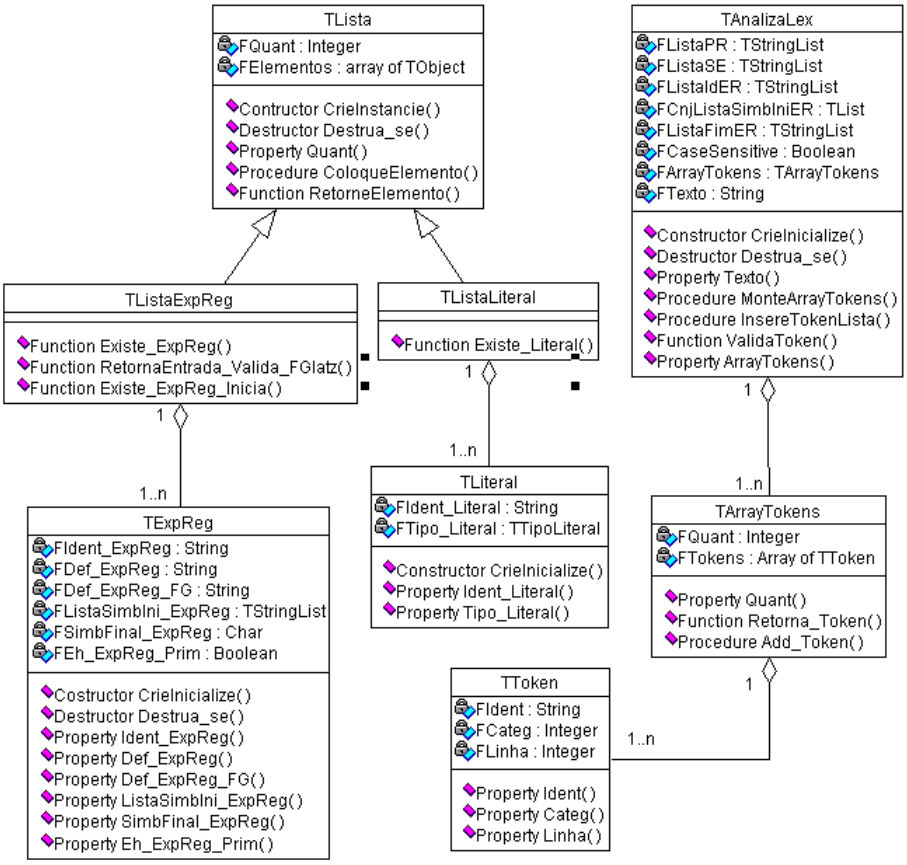
No quadro 3.5 tem-se o segundo *use case*, que descreve como é utilizada a classe gerada.

QUADRO 3.5 – *USE CASE:* UTILIZAR A CLASSE GERADA

<table>
<tr><td colspan="2"> 
</td></tr>
<tr><td>USE CASE:</td><td>Utilizar classe gerada</td></tr>
<tr><td>BREVE DESCRIÇÃO:</td><td>Em uma janela do protótipo, o usuário irá validar a especificação feita, testando a classe gerada, sem precisar compilar tal classe.</td></tr>
<tr><td>ATOR(ES):</td><td>Usuário</td></tr>
<tr><td>PRÉ-CONDIÇÕES:</td><td>É necessário que uma classe tenha sido gerada para uma especificação de entrada, ou seja, não deve ter ocorrido erro na análise da especificação.</td></tr>
<tr><td>FLUXO PRINCIPAL:</td><td>1. O usuário, de acordo com a especificação de entrada, entra com algumas palavras reservadas, símbolos especiais e outros tokens para serem analisados e seleciona o botão Analisar.
2. O protótipo compila e executa o código fonte que é escrito em tempo de execução, e que simula a classe gerada para fazer a análise léxica.
3. A classe gerada identifica os tokens, um a um, mostrando o resultado obtido da análise, ou seja, cada token reconhecido com sua respectiva categoria bem como a linha em que o mesmo se encontra.</td></tr>
<tr><td>FLUXOS ALTERNATIVOS E EXCEÇÕES:</td><td>Caso algum token não seja reconhecido, será mostrado na saída o token, a categoria zero (0), a descrição de token inválido e a linha onde o erro foi identificado.</td></tr>
<tr><td>PÓS-CONDIÇÕES:</td><td>O teste da classe gerada foi feito.</td></tr>
<tr><td>REQUISITOS FUNCIONAIS SATISFEITOS:</td><td>O protótipo deve permitir de algum modo testar a classe gerada.</td></tr>
</table>

Para a implementação do protótipo, foi utilizada a ferramenta geradora de compiladores *Coco/R for Delphi*, que será descrita na seção 3.3.1. Esta ferramenta gera automaticamente as análises léxica e sintática, ou seja, deve-se implementar somente a análise semântica. Por isso, a especificação das classes para as análises léxica e sintática não será descrita, sendo mostrado apenas o código de entrada do *Coco/R for Delphi* onde são especificadas tais análises. Para fazer a análise semântica da entrada foram criadas algumas classes, conforme o diagrama de classe mostrado na fig. 3.2.

FIGURA 3.2 – DIAGRAMA DE CLASSES



A classe *TLista* define dois atributos, sendo que um indica a quantidade de elementos armazenados na lista e o outro é uma lista de objetos do tipo *TObject*. Esta classe possui métodos públicos para inserir os elementos na lista (*ColoqueElemento*), para ler e escrever a propriedade *FQuant*, que representa a quantidade de elementos na lista (property *Quant*), para retornar o elemento de uma certa posição da lista (*RetorneElemento*), além dos métodos para criar e para destruir um objeto dessa classe (*CrieInstancia* e *Destrua_se*).

A classe *TLista* é superclasse das classes *TListaLiteral* e *TListaExpReg*, as quais implementam listas específicas de literais e de expressões regulares, respectivamente, e além de herdarem as propriedades e métodos da superclasse *TLista*, possuem seus próprios métodos. A classe *TListaLiteral* possui um único método público que verifica a existência de um objeto na lista (*Existe_Literal*) e a classe *TListaExpReg* possue 3 métodos públicos, onde um verifica a existência de um objeto na lista (*Existe_ExpReg*), outro lê todos os elementos da lista e retorna um único *string* contendo todas as definições regulares para geração do AFD (*RetornaEntrada_Valida_FGlatz*) e o último método verifica a existência de uma ER na lista

que têm como possíveis símbolos iniciais algum dos símbolos passados por parâmetro (*Existe_ExpReg_Inicia*).

Os elementos da lista de literais são objetos da classe *TLiteral*, que podem ser as palavras reservadas ou os símbolos especiais da linguagem que está sendo definida, sendo diferenciados por uma propriedade que diz se o literal é uma palavra reservada ou símbolo especial. A classe *TLiteral* possui 3 métodos públicos, sendo um para criar uma instância do objeto e outros dois (*property)* que permitem a escrita e a leitura das propriedades *FIdent_Literal* e *FTipoLiteral*, denominados *Ident_Literal* e *TipoLiteral*, respectivamente.

Já a lista de expressões regulares tem como elementos instâncias da classe *TExpReg*, isto é, expressões regulares. Um objeto dessa classe tem como atributos um identificador para a expressão regular (*FIdent_ExpReg*), a expressão em si (*FDef_ExpReg*), a expressão na forma de entrada para a implementação de Glatz (2000) (*FDef_ExpReg_FG*), uma lista de possíveis símbolos iniciais (*FListaSimbIni_ExpReg*), um caracter com o símbolo que a expressão regular deve finalizar, se for o caso (FSimbFinal_ExpReg), e um atributo que diz se a expressão é ou não primária (*FEh_ExpReg_Prim*), ou seja, se for primária somente serve para definir outros *tokens*, caso contrário é um *token* final que deverá ser reconhecido como unidade básica da linguagem definida. A classe *TExpReg* possui seis métodos públicos (*property*), que possibilitam a escrita e a leitura de propriedades, quais sejam *Ident_ExpReg, Def_ExpReg, Def_ExpReg_FG, FListaSimbIni_ExpReg, FSimbFinal_ExpReg e FEh_ExpReg_Prim.* Também possui um método para criar e inicializar uma instância da classe (*CrieInicialize*).

A classe final, conforme visto na fig. 3.1, denomina-se *TAnalisaLex* e possui em sua definição outras duas classes denominadas *TArrayTokens* e *TToken*, necessitando destas outras duas classes para ser utilizada. A classe gerada *TAnalisaLex* possui as seguintes propriedades e métodos:

a) FListaPR: lista de palavras reservadas;
b) FListaIdER: lista com os identificadores de *tokens* que serão reconhecidos, ou seja, os identificadores das ER que não são primárias conforme descrito acima;
c) FCnjListaSimbIniER: lista com todos os símbolos que os *tokens* podem iniciar;
d) FListaFimER: lista com os símbolos finais das ER;

e) FCaseSensitive: valor lógico que indica se foi informado “CAIXA SENSIVEL” na especificação de entrada do protótipo;
f) FArrayTokens: *array* dinâmico do tipo *TArrayTokens* para armazenar objetos da classe *TToken*, que serão os *tokens* lidos, reconhecidos e classificados;
g) FTexto: *string* que conterá o texto a ser analisado.

Possui os seguintes métodos privados:

a) *InsereTokenLista*, que recebe como parâmetros um *string*, ou seja, o *token* reconhecido, e um inteiro que indica o número da linha onde o mesmo foi encontrado, e insere um *TToken* em *FArrayTokens*;
b) *ValidaToken*, que implementa um AFD para reconhecer ou não os *tokens* passados um a um como parâmetro.

Possui os seguintes métodos públicos:

a) *MontaArrayTokens*, que lê todo o texto caracter a caracter, separando em cadeias que poderão ser classificadas como símbolos especiais, palavras reservadas ou reconhecidas pelo método *ValidaToken*. Após passarem por essa verificação, cada cadeia com sua categoria forma um objeto *TToken*, que é armazenado em *FarrayTokens* através do método *InsereTokenLista*;
b) *CrieInicialize*, que cria uma instância do objeto *TAnalisaLex* e inicializa todas as suas propriedades;
c) *Destrua_se*, que destrói a instância desalocando o espaço reservado de memória;
d) *Texto* (*property*), que lê e escreve o valor da propriedade *FTexto*;
e) *ArrayTokens* (*property*), que lê e escreve os valores da propriedade *FArrayTpkens*, encapsulando todas as funcionalidades de um *TArrayTokens*.

Como já visto anteriormente, a classe *TArrayTokens* implementa uma lista cujos elementos são objetos *TToken*, possuindo uma propriedade *FTokens*, que é um *array* dinâmico de *Ttoken*, e *FQuant*, que guarda a quantidade de objetos da lista. Nesta classe existe um método (*property)* que permite a leitura e a escrita na propriedade *FQuant* denominado *Quant*, um método que adiciona e outro que retorna um objeto *TToken*, denominados *Add_Token* e *Return_Token*, respectivamente.

A última classe do protótipo a ser descrita é a classe *TToken*, que possui 3 propriedades denominadas *FIdent, FCateg* e *FLinha*, onde para cada uma das propriedades existe um método (property) que permite sua escrita e leitura.

3.3 IMPLEMENTAÇÃO

3.3.1 TÉCNICAS E FERRAMENTAS UTILIZADAS

Este protótipo foi implementado na linguagem Object Pascal no ambiente Delphi 5 (Cantù, 2000). Para gerar o compilador da meta-linguagem foi utilizado o *CocoR for Delphi*. Para transformar uma expressão regular em um autômato finito determinístico foi utilizado o algoritmo descrito por Silva (2000). Para utilizar a classe gerada, foram usados os componentes de *script* IFPS3 (*INNERFUSE PASCAL SCRIPT III*).

3.3.1.1 *COCO/R FOR DELPHI*

Conforme descrito na introdução, *Coco/R for Delphi*, de acordo com Reith (2002), é um compilador de compilador *freeware* para gerar em Delphi um analisador sintático recursivo descendente LL(1) constituído ou por classes de objetos ou por projetos completos que incluem dois componentes *TRichText*, sendo um para entrada e outro para saída, de um aplicativo que pode ser utilizado para testar as classes geradas.

A entrada para *Coco/R for Delphi* é um arquivo com a extensão *ATG* contendo a especificação de uma linguagem feita em termos de definições regulares e de uma gramática livre de contexto. Nesse arquivo também pode ser incluído código Delphi para programar a semântica da linguagem especificada. No arquivo *MetaLinguagem.ATG* (Apêndice 1), criado na implementação deste protótipo, estão as definições regulares, a especificação da sintaxe e da semântica da meta-linguagem. Os quadros 3.5 e 3.6 mostram, respectivamente, uma especificação parcial das definições regulares e da sintaxe e da semântica da meta-linguagem, descritas de acordo com a notação aceita pela ferramenta *Coco/R for Delphi* e em conformidade com a especificação feita na seção 3.2.1.

QUADRO 3.6 – DEFINIÇÕES REGULARES DA META-LINGUAGEM NO *COCO/R FOR DELPHI*

```
IGNORE CASE

CHARACTERS
  Maius   = CHR(65)..CHR(90) .
  Minus   = CHR(97)..CHR(122) .
  Digit   = CHR(48)..CHR(57) .
  Aspa    = CHR(39).
  Aspas   = CHR(34).
  EOL     = CHR(13).
  N1      = CHR(38).  // &
  N2      = CHR(40).  // (
  N3      = CHR(41).  // )
  N4      = CHR(42).  // *
  N5      = CHR(94).  // ^
  N6      = CHR(124). // |
  ASCII   = ANY - EOL.
  Def_Lit = ASCII - Aspas - N1 - N2 - N3 - N4 - N5 - N6.
  Def_Car = ASCII - N1 - N2 - N3 - N4 - N5 - N6.

TOKENS
  Ident     = Maius | Minus { "_" | Maius | Minus | Digit}.
  Literal   = Aspas Def_Lit {Def_Lit} Aspas .
  Caracter = Aspa Def_Car Aspa.

COMMENTS FROM "//" TO EOL
COMMENTS FROM "/*" TO "*/" NESTED

IGNORE CHR(1)..CHR(32)
```

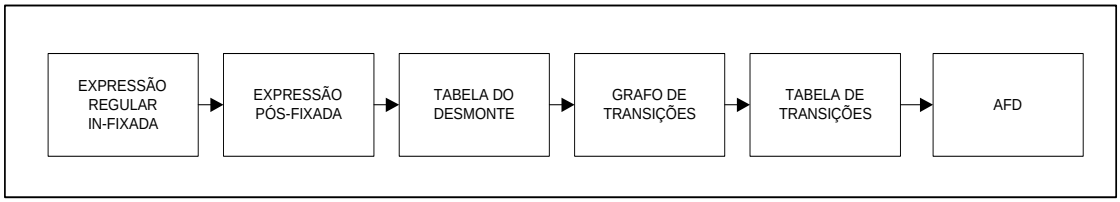
QUADRO 3.7 – SINTAXE DA META-LINGUAGEM NO *COCO/R*

```
Meta_Linguagem =                        (. Inicializar; .)
                 "PALAVRAS"
                 "RESERVADAS"
                 ":"
                 { "("
                   "CAIXA"
                   "SENSIVEL"
                   ")"                  (. CaseSensitive := True; .)
                 }
                 "{"                    (. TipoLiteral := tPR; .)
                 [Cnj_Literal]
                 "}"
                 "SIMBOLOS"
                 "ESPECIAIS"
                 ":"
                 "{"                    (. TipoLiteral := tSE; .)
                 [Cnj_Literal]
                 "}"
                 "TOKENS"
                 ":"
                 "{"
                 Cnj_Token
                 "}"
                                        (. Finalizar; .)
(...)
```

3.3.1.2 ALGORITMO DESCRITO POR SILVA (2000)

A partir da compilação da especificação da linguagem, informada na entrada do protótipo, tem-se uma lista de expressões regulares transformadas em entradas válidas para o "Desmonte", que é o nome dado ao algoritmo descrito por Silva (2000) e implementado por Glatz (2000), e que transforma as ER em um único AFD. Segundo Glatz (2000), as etapas para realizar a transformação de uma ER em um AFD através do algoritmo descrito por Silva (2000) podem ser observadas na fig.3.3.

FIGURA 3.3 – ETAPAS DO ALGORITMO DESCRITO POR SILVA (2000)



Fonte: Glatz (2000)

Glatz (2000) apresenta em detalhes esse algoritmo. Resumidamente, tem-se cinco etapas. A primeira etapa é, a partir da expressão regular de entrada, obter uma expressão pós-fixada. Em seguida, é construída uma "tabela de desmonte" através de uma série de passos. A partir da "tabela de desmonte" é criada uma outra tabela que implementa o grafo de transição correspondente. A última etapa cria, a partir da tabela criada a partir da "tabela de desmonte", a tabela de transição que implementa um autômato finito determinístico.

3.3.1.3 *INNERFUSE PASCAL SCRIPT III* (IFPS3)

Para utilizar a classe gerada, ou melhor, simular a classe conforme será descrito abaixo, foram usados os componentes de *script* IFPS3. Tais componentes possibilitam compilar e executar em um programa Delphi em tempo de execução, códigos fonte de acordo com a sintaxe da linguagem Object Pascal. A única restrição encontrada para tais códigos fonte é não poder declarar uma classe (*Class*), sendo possível somente utilizar uma classe já criada, ou qualquer classe da *Visual Component Library* (VCL). Para isto, são necessários dois componentes: *TIFPS3ClassesPlugin* e *TIFPS3CompExec*, onde o primeiro é utilizado somente para registrar as classes que serão utilizadas pelo código fonte, e o segundo para

registrar os objetos que serão instanciados, chamar o método compilar e posteriormente o método executar.

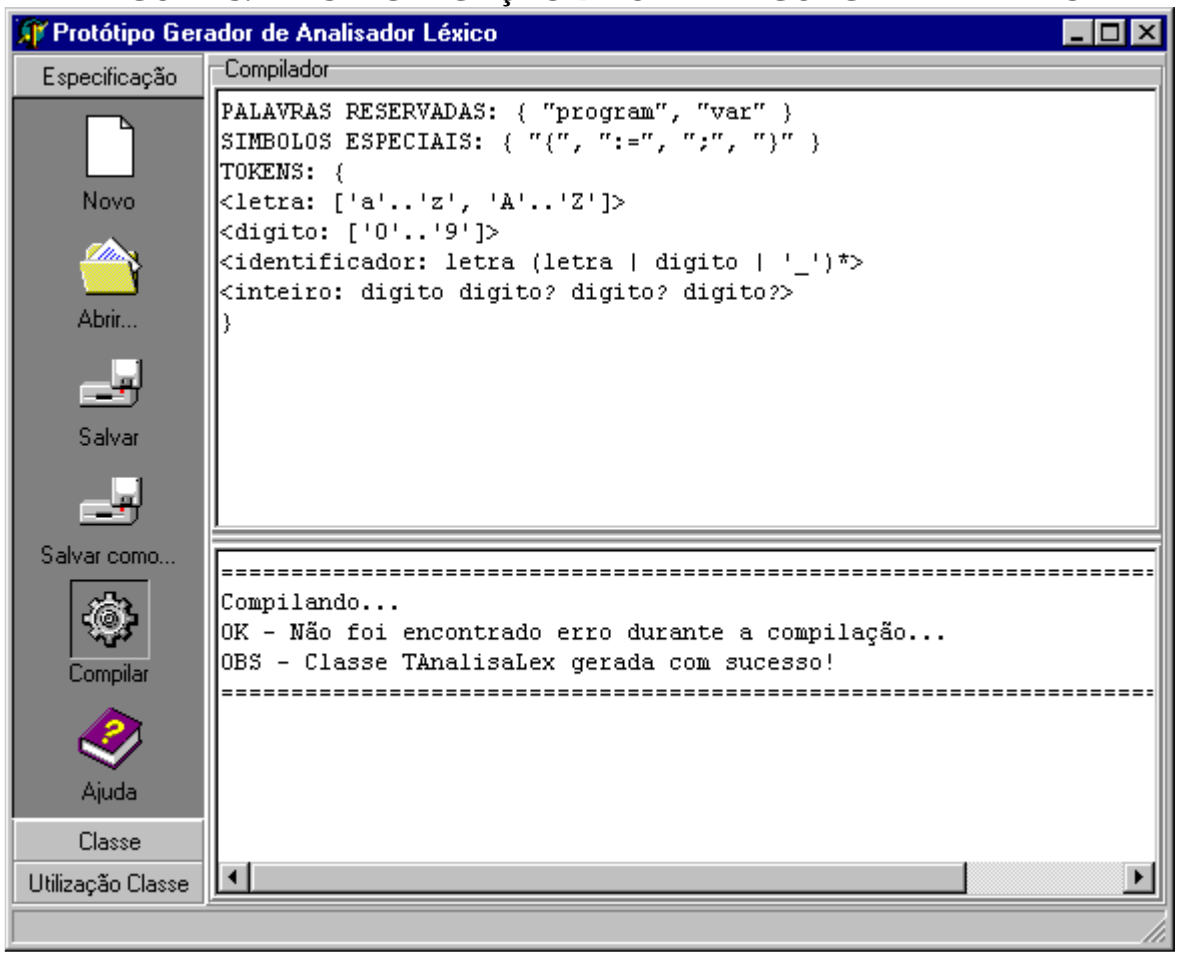
Assim, como não se pode utilizar uma classe criada em tempo de execução, para realizar o teste da classe *TAnalisaLex*, foi feita uma simulação da mesma, de maneira que suas características e funcionalidades foram mantidas. Todo o código fonte desta classe é escrito em um campo que fica invisível no protótipo, sendo seus métodos declarados como *functions* ou *procedures* e suas propriedades como variáveis. Isto é, este código é uma cópia da classe gerada, disponibilizado de tal forma que a classe *TIFPS3CompExec* recebe o código fonte, compila e executa, através de métodos distintos que são chamados explicitamente e que pertencem a própria classe.

A principal dificuldade em utilizar tais componentes é a falta de exemplos demonstrativos de suas funcionalidades, além de conter uma documentação insuficiente dos componentes citados.

3.3.2 OPERACIONALIDADE DA IMPLEMENTAÇÃO

O protótipo desenvolvido apresenta uma interface amigável, onde todas as suas funcionalidades são encontradas em uma barra de ferramentas vertical ao lado esquerdo do aplicativo, conforme pode ser visto nas figuras 3.4, 3.5 e 3.6.

FIGURA 3.4 – ESPECIFICAÇÃO DE UMA LINGUAGEM EXEMPLO



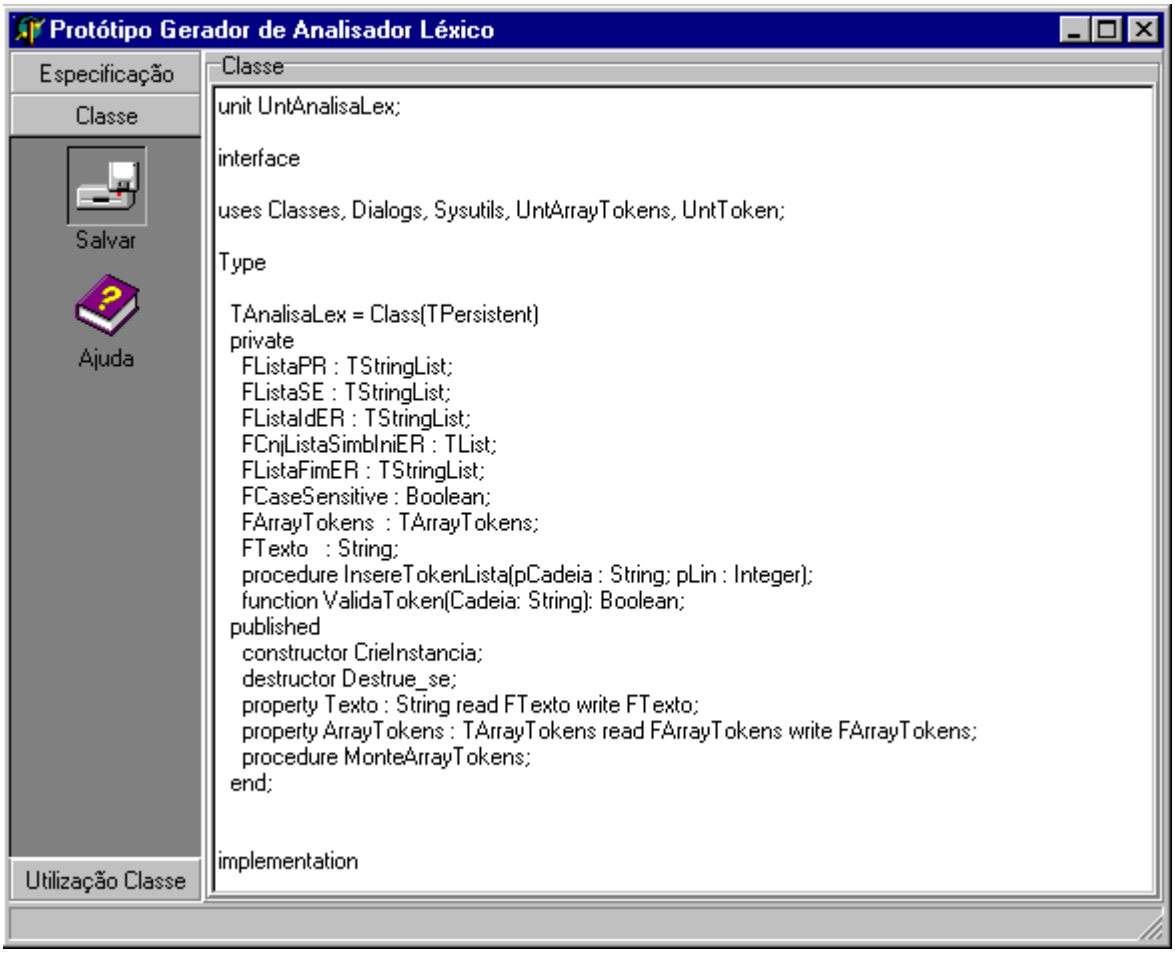
A interface do protótipo foi dividida em três partes destacáveis, descritas por “Especificação”, “Classe” e “Utilização Classe”, as quais são chamadas de itens, sendo que dentro de cada um destes itens existem outros sub-itens, detalhados a seguir.

O primeiro item é utilizado quando o usuário deseja fazer a especificação dos *tokens* de uma nova linguagem de programação. Conforme é visto na fig. 3.4, o item “Especificação” possui dois componentes de textos, um servindo para a definição dos *tokens*, de acordo com a meta-linguagem especificada, e outro usado para mostrar a saída da compilação, ou seja, uma mensagem indicando que a especificação foi compilada com sucesso ou erros léxicos, sintáticos e semânticos identificados. Esse item possui seis sub-itens na barra de ferramentas, denominados “Novo”, “Abrir...”, “Salvar”, “Salvar como...”, “Compilar” e “Ajuda”, onde “Novo” abre um novo arquivo em branco; “Abrir” abre um arquivo contendo uma especificação de entrada; “Salvar” permite salvar a especificação de entrada; “Salvar como”

permite salvar o arquivo com um outro nome; "Compilar" analisa o texto de entrada e, se não houver erros, gera automaticamente a classe *TAnalisaLex* de acordo com a definição apresentada na fig. 3.1; e "Ajuda" apresenta um *help* contendo explicações sobre o item e seus sub-itens.

A classe gerada na compilação poderá ser vista no item "Classe", conforme é mostrado na fig. 3.5. O código fonte completo da classe gerada para a especificação exemplo da fig. 3.4 encontra-se no Anexo 2.

FIGURA 3.5 – CLASSE GERADA

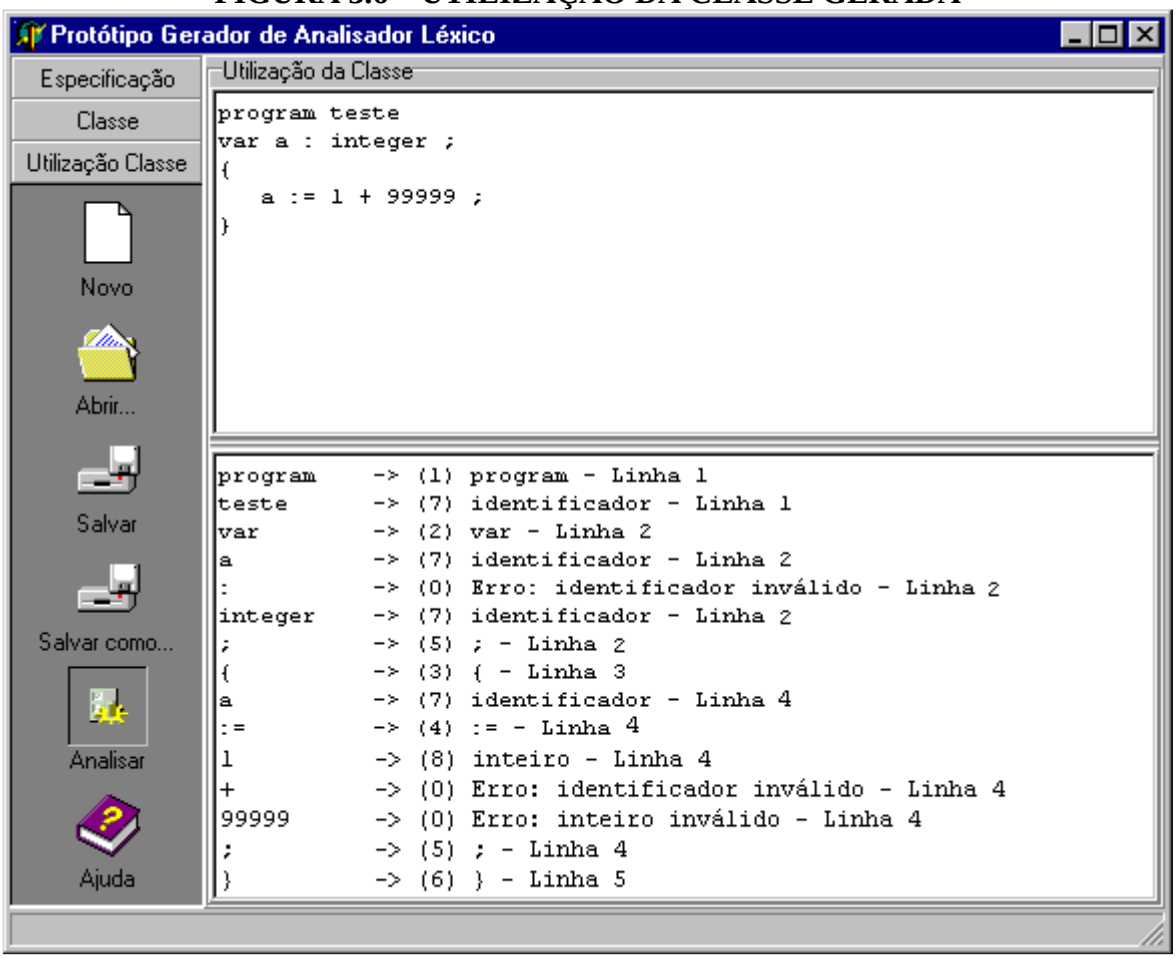


Neste item é mostrada a classe que implementa o analisador léxico para a especificação de entrada, objetivo final deste protótipo, de acordo com a sintaxe aceita pela linguagem Object Pascal. Possui dois sub-itens na barra de ferramentas, denominados

"Salvar" e "Ajuda", onde "Salvar" permite salvar a classe gerada em um arquivo de extensão PAS e "Ajuda" apresenta um *help* contendo explicações sobre a classe e suas funcionalidades.

A classe gerada *TAnalisaLex* pode ser testada no item "Utilização Classe", conforme é visto na fig. 3.6. Para a demonstração da utilização da classe, são utilizados dois componentes de textos, um para a entrada do programa fonte e outro para a saída do analisador léxico. O texto de entrada será formado por palavras reservadas, símbolos especiais e outros *tokens*, de acordo com o padrão de formação definido. Na saída serão listados os *tokens* reconhecidos, suas categorias e o número das linhas onde se encontram. Se houver erro no reconhecimento de um *token*, será mostrada uma mensagem de erro. Neste item são encontrados os sub-itens "Novo" , "Abrir...", "Salvar", "Salvar como...", "Analisar" e "Ajuda", onde os sub-itens "Novo" , "Abrir...", "Salvar", "Salvar como..." e "Ajuda" têm a mesma funcionalidade descrita anteriormente, e o sub-item "Analisar" executa a análise léxica do texto de entrada.

FIGURA 3.6 – UTILIZAÇÃO DA CLASSE GERADA



As categorias mostradas na fig. 3.6 entre parênteses são enumeradas dinamicamente pela classe, conforme a ordem de entrada da especificação, onde são primeiramente enumeradas as palavras reservadas, posteriormente os símbolos especiais e por último os outros *tokens*, definidos pelas expressões regulares não primárias. Como pode-se observar, a categoria zero “(0)” é utilizada quando o *token* não é reconhecido.

4 CONCLUSÕES

Foram vistos neste trabalho diversos conceitos, tais como definição de compiladores, ferramentas geradoras de compiladores e especificações formais das análises léxica e sintática. A teoria estudada é de extrema importância para o desenvolvimento de um protótipo de gerador de analisador léxico, que foi implementado com êxito, alcançando o objetivo proposto, qual seja gerar uma classe que implementa um analisador léxico juntamente com outras duas classes, a partir de uma especificação, abstraindo a complexidade dos algoritmos utilizados.

O protótipo implementado possui uma interface de fácil manipulação. Poderá ser utilizado por alunos da disciplina de Compiladores ou por qualquer pessoa que queira gerar um analisador léxico em Object Pascal. É uma ferramenta *freeware* de fácil utilização com uma pequena limitação no alfabeto de entrada das expressões regulares, que não pode conter caracteres do conjunto {‘&’, ‘|’, ‘*’, ‘^’. ‘(’,’)’}, os quais são utilizados como operadores na implementação do algoritmo descrito por Silva (2000). Outra restrição diz respeito aos símbolos especiais que devem ser formados por no máximo dois caracteres.

A utilização da classe gerada *TAnalisaLex,* juntamente com as classes *TArrayTokens* e *TToken* é muito simples, sendo necessário apenas adicionar suas respectivas *units* *UntAnalisaLex.pas*, *UntArrayTokens.pas* e *UntToken.pas* a um projeto Delphi 4, ou versão superior, pelo motivo de conter um *array* dinâmico na classe *TArrayTokens*. Após adicioná-las à cláusula *uses*, pode-se declarar um objeto do tipo *TAnalisaLex*. Deverá ser criada uma instância deste objeto, atribuído um texto fonte à propriedade *Texto* e chamado o método *MonteArrayTokens*, que monta uma lista interna da classe com todos os *tokens* existentes no texto fonte de entrada. Então, é só varrer a lista de *tokens* mostrando-os em um texto de saída, ou passá-los à próxima fase do compilador, a análise sintática.

4.1 EXTENSÕES

Pode-se deixar como sugestões para trabalhos futuros uma melhoria no protótipo, onde deixariam de existir as restrições descritas na conclusão, deste modo, o alfabeto de entrada das expressões regulares poderá conter os caracteres do conjunto {‘&’, ‘|’, ‘*’, ‘^’. ‘(’,’)’} e os símbolos especiais poderão ser formados por mais de dois caracteres.

Outra sugestão para trabalho futuro é a implementação de um gerador de analisador sintático e mais adiante a implementação de um gerador de gerador de código objeto.

APÊNDICE 1 - METALINGUAGEM.ATG

A seguir encontra-se o arquivo metalinguagem.atg criado na implementação deste protótipo.

A parte de código escrita entre *(* Inicio - Código Form *) e (* Fim - Código Form *)*, serve para especificar o código fonte do *form* (*TForm*) que será gerado pelo *Coco/R for Delphi*, além de suas propriedades e métodos, que serão utilizados nas análises sintática e semântica.

No código escrito entre *(* Inicio - Especificação Léxica *) e (* Fim - Especificação Léxica *)* está a especificação dos *tokens* da meta-linguagem. Contém as seguintes cláusulas: IGNORE CASE, CHARACTERS, TOKENS, COMMENTS e IGNORE CHR, onde IGNORE CASE especifica que não será feita distinção entre letras maiúsculas e minúsculas; CHARACTERS define os caracters ou conjunto de caracters que serão utilizados na cláusula TOKENS, onde são especificados as definições regulares; COMMENTS define como serão os comentários de linha ou de bloco; e IGNORE CHR define um conjunto de caracteres que não serão reconhecidos pela análise léxica.

Na última parte do arquivo, encontra-se o código fonte escrito entre *(* Inicio - Especificação Sintática *) e (* Fim - Especificação Sintática *)*, onde é especificada a sintaxe da meta-linguagem juntamente com os códigos que implementam a análise semântica, escritos em uma coluna à direita entre as definições “(.” e “.)”.

```
COMPILER Meta_Linguagem

(* Inicio - Código Form *)

DELPHI
  USES (INTERFACE)
       UntListaLiteral, UntLiteral, UntListaExpReg, UntExpReg
  TYPE
    PRecordER = ^RecordER;
    RecordER = record
       IdentER : String;
       DefER   : String;
    end;
  PRIVATE
    fOutputStream : TMemoryStream;
    ListaPR_Dup : TList;
    ListaSE_Dup : TList;
    ListaER_Dup : TList;
    ListaCaracter : TList;
    ListaSimbIniER : TStringList;
    PString : ^String;
    PChar   : ^Char;
```

```
    RegER   : PRecordER;
    procedure Inicializar;
    procedure Finalizar;
    procedure EscreveStr(S : string);
    procedure InicializaListas;
    procedure InsereListaLiteral;
    procedure MontaFaixa_FG;
    procedure SomaER(pTipo : Integer);
    procedure ConsisteIdentER;
    function  ConsisteIdentER_NaDefinicao : Integer;
    procedure InsereListaER;
    procedure InsereListaPR_Dup(pStrPRDup : String);
    procedure InsereListaSE_Dup(pStrSEDup : String);
    procedure InsereListaER_Dup(pIdentExpressao, pDefExpressao : String);
    procedure InsereListaCaracter(Caracter : Char);
    function  RetornaUltPR_Dup : String;
    function  RetornaUltSE_Dup : String;
    function  RetornaUltER_Dup(IndiceCampo : Integer) : String;
    function  RetornaCaracter : String;
    procedure EscreveUltimoCaracter;
    procedure ConsisteCaracterFaixa(OrdemCaracter : Integer);
    property OutputStream : TMemoryStream read fOutputStream write fOutputStream;
  PUBLIC
    vListaLiteral : TListaLiteral;
    vListaExpReg : TListaExpReg;
  CREATE
    fOutputStream := TMemoryStream.Create;
    ListaPR_Dup := TList.Create;
    ListaSE_Dup := TList.Create;
    ListaER_Dup := TList.Create;
    ListaCaracter := TList.Create;
    ListaSimbIniER := TStringList.Create;
    CnjTodos := TStringList.Create;
    Faixa_FG := TStringList.Create;
    PosCnj := 1;
    CnjTodos.Clear;
    for PosCar := 33 to 126 do
      if not (PosCar in [38,40,41,42,94,124]) then
        CnjTodos.Add(CHR(PosCar));
  DESTROY
    fOutputStream.Free;
    vListaLiteral.Free;
    vListaExpReg.Free;
    ListaPR_Dup.Free;
    ListaSE_Dup.Free;
    ListaER_Dup.Free;
    ListaCaracter.Free;
    ListaSimbIniER.Free;
    CnjTodos.Free;
    Faixa_FG.Free;
  ERRORS
    200 : Result := 'Palavra reservada '+RetornaUltPR_Dup+' já encontrada';
    201 : Result := 'Símbolo especial '+RetornaUltSE_Dup+' já encontrado';
    202 : Result := 'Identificador de token "'+RetornaUltER_Dup(0)+'" já encontrado';
    203 : Result := 'Expressão regular "'+RetornaUltER_Dup(1)+'" já encontrada';
    204 : Result := 'O caracter '+RetornaCaracter+' já existe na faixa';
    205 : begin
            StrErroAux := RetornaCaracter;
            Result := 'O caracter '+RetornaCaracter+' deve ser maior que o caracter
'+StrErroAux;
          end;
    206 : begin
            StrErroAux := RetornaCaracter;
            Result := 'O caracter '+RetornaCaracter+' deve ser diferente do caracter
'+StrErroAux;
          end;
    207 : Result := 'O caracter '+RetornaCaracter+' contido nesta sub-faixa já existe na
faixa';
    208 : Result := 'Expressão regular "'+RetornaUltER_Dup(0)+'" não encontrada
anteriormente';
    209 : Result := 'O símbolo especial '+RetornaUltSE_Dup+' não pode conter mais que dois
caracteres';
    210 : begin
            StrErroAux := RetornaUltER_Dup(0);
```

```
            StrErroAux := StrErroAux + ' : ' + RetornaUltER_Dup(1);
            Result := 'A expressão "'+RetornaUltER_Dup(0)+' : '+RetornaUltER_Dup(1)+'" não
pode ter como possível símbolo inicial'+#13+
                      ' um já definido em uma expressão anteriormente, no caso:
"'+StrErroAux+'';
          end;
END_DELPHI

const
  PadLen = 20;

type
  TCnjCaraterASCII  = Set of Byte;

var
  TipoLiteral     : TTipoLiteral;
  IdentExpressao  : String;
  DefExpressao    : String;
  CnjCaraterASCII : TCnjCaraterASCII;
  Caracter1       : Char;
  Caracter2       : Char;
  StrErroAux      : String;
  AchouCaracter1  : Boolean;
  CaseSensitive   : Boolean;

  DefExpressao_FG : String;
  CnjTodos        : TStringList;
  Faixa_FG        : TStringList;
  PosER           : Integer;
  PosCar          : Integer;
  PosCnj          : Integer;
  UltCarac4       : String;
  OperNeg         : Boolean;

  Nivel           : Integer;
  SimbFimER       : Char;


(*******Insere na Lista de Expressões Regulares Duplicadas*******)
procedure T-->Grammar<--.InsereListaER_Dup(pIdentExpressao, pDefExpressao : String);
begin
  New(RegER);
  RegER^.IdentER := pIdentExpressao;
  RegER^.DefER   := pDefExpressao;
  ListaER_Dup.Add(RegER);
end;


(*******Inicialização para a Análise da GLC*******)
procedure T-->Grammar<--.Inicializar;
begin
  InicializaListas;
  IdentExpressao := '';
  DefExpressao := '';
  DefExpressao_FG := '';
  CaseSensitive := False;
  OutputStream.Clear;

EscreveStr('================================================================================
=');
  EscreveStr('Compilando...');
end;

(*******Finalização da Análise da GLC*******)
procedure T-->Grammar<--.Finalizar;
begin
  if ErrorList.Count = 0 then
    begin
    EscreveStr('OK - Não foi encontrado erro durante a compilação... ');
    EscreveStr('OBS - Classe TAnalisaLex gerada com sucesso!');
    end
  else
    begin
    if ErrorList.Count = 1 then
```

```
      EscreveStr('ERRO - Foi encontrado 1 erro durante a compilação...')
    else
      EscreveStr('ERRO - Foram encontrados '+IntToStr(ErrorList.Count)+' erros durante a
compilação...');
    EscreveStr('OBS - A Classe não pode ser gerada!');
    end;


EscreveStr('================================================================================
=');
  OutputStream.Position := 0;
  OutputStream.SaveToStream(ListStream);
  OutputStream.Clear;
end;

(*******Escreve nova linha no Texto de Resultado da Análise*******)
procedure T->Grammar<-.EscreveStr(S : string);
begin
  S := S + #13#10;
  OutputStream.WriteBuffer(S[1],length(S));
end;

(*******Inicializa as Listas de PR, e ER*******)
procedure T->Grammar<-.InicializaListas;
var
  x : Byte;
begin
{Libera as instancias da classe TLista e instancia novamente}
  vListaLiteral.Free;
  vListaLiteral := TListaLiteral.CrieInstancia;
  vListaExpReg.Free;
  vListaExpReg := TListaExpReg.CrieInstancia;

{Limpa a lista de Palavras Reservadas Duplicadas}
  if ListaPR_Dup.Count > 0 then
    for x := 0 to (ListaPR_Dup.Count - 1) do
      begin
        PString := ListaPR_Dup.Items[x];
        Dispose(PString);
      end;
{Limpa a lista de Símbolos Especiais Duplicados}
  if ListaSE_Dup.Count > 0 then
    for x := 0 to (ListaSE_Dup.Count - 1) do
      begin
        PString := ListaSE_Dup.Items[x];
        Dispose(PString);
      end;
{Limpa a lista de Expressões Regulares Duplicadas}
  if ListaER_Dup.Count > 0 then
    for x := 0 to (ListaER_Dup.Count - 1) do
      begin
        RegER := ListaER_Dup.Items[x];
        Dispose(RegER);
      end;
{Limpa a lista de Caracteres das Faixas com erro}
  if ListaCaracter.Count > 0 then
    for x := 0 to (ListaCaracter.Count - 1) do
      begin
        PChar := ListaCaracter.Items[x];
        Dispose(PChar);
      end;

  ListaPR_Dup.Clear;
  ListaSE_Dup.Clear;
  ListaER_Dup.Clear;
  ListaCaracter.Clear;
  ListaSimbIniER.Clear;
end;

(*******Chama procedimento para Inserir PR ou SE em TListaLiteral*******)
procedure T->Grammar<-.InsereListaLiteral;
var
  vLiteral : TLiteral;
begin
```

```
  if not vListaLiteral.Existe_Literal(Copy(LexString,2,Length(LexString)-2), TipoLiteral)
then
    begin
      if (TipoLiteral = tSE)and(length(LexString) > 4) then
        begin
        InsereListaSE_Dup(LexString);
        SynError(209);
        end
      else
        begin
        vLiteral := TLiteral.CrieInicialize(Copy(LexString,2,Length(LexString)-2),
TipoLiteral);
        vListaLiteral. coloqueElemento(vLiteral)
        end;
    end
  else
    if TipoLiteral = tPR then
      begin
        InsereListaPR_Dup(LexString);
        SynError(200);
      end
    else
      begin
        InsereListaSE_Dup(LexString);
        SynError(201);
      end;
end;

(*******Soma os tokens que formarão a Expressão Regular*******)
procedure T-->Grammar<--.MontaFaixa_FG;
var
  i,j : Integer;
  Pipe : String;
begin
  DefExpressao_FG := DefExpressao_FG + '(';
  Pipe := '';
  if OperNeg then
    begin
      for  i := 0 to CnjTodos.Count - 1 do
        begin
          j := Faixa_FG.IndexOf(CnjTodos[i]);
          {Se nao encontrou o caracter na lista "Faixa_FG"}
          if j < 0 then
            begin
              DefExpressao_FG := DefExpressao_FG + Pipe + CnjTodos[i];
              Pipe := '|';
              if Nivel = 1 then
                ListaSimbIniER.Add(CnjTodos[i]);
             end;
        end;
    end
  else
    begin
      for  i := 0 to Faixa_FG.Count - 1 do
        begin
          DefExpressao_FG := DefExpressao_FG + Pipe + Faixa_FG[i];
          Pipe := '|';
          if Nivel = 1 then
            ListaSimbIniER.Add(Faixa_FG[i]);
        end;
    end;
  DefExpressao_FG := DefExpressao_FG + ')';
end;

(*******Soma os tokens que formarão a Expressão Regular*******)
procedure T-->Grammar<--.SomaER(pTipo : Integer);
var
   sAux : String;
   i, PAbre, PFecha : Integer;
   Achou : Boolean;
begin
  if pTipo in [1] then
    DefExpressao := DefExpressao + ' ' + LexString
  else
```

```
    DefExpressao := DefExpressao + LexString;

  case pTipo of
    1 : //Somar normalmente
        DefExpressao_FG := DefExpressao_FG + LexString;
    2 : //Somar Caracter retirando as aspas
        begin
          sAux := Copy(LexString, 2, 1);
          DefExpressao_FG := DefExpressao_FG + sAux;
        end;
    3 : //Somar propriedade de uma outra ER da Lista posicionada em PosER
        if PosER >= 0 then
          begin
            sAux := TExpReg(vListaExpReg.Return_Elemento(PosER+1)).Def_ExpReg_FG;
            DefExpressao_FG := DefExpressao_FG + '(' + sAux + ')';
            if Nivel = 1 then
              for i := 0 to
TExpReg(vListaExpReg.Return_Elemento(PosER+1)).ListaSimbIni_ExpReg.Count - 1 do

ListaSimbIniER.Add(TExpReg(vListaExpReg.Return_Elemento(PosER+1)).ListaSimbIni_ExpReg[i]);
          end;
    4 : //Somar Faixa de Caracteres - 1º Caracter
        begin
          sAux := Copy(LexString, 2, 1);
          UltCarac4 := sAux;
          Faixa_FG.Add(sAux);
        end;
    5 : //Somar Faixa de Caracteres - 2º Caracter
        begin
          sAux := Copy(LexString, 2, 1);
          for i := Ord(UltCarac4[1])+1 to Ord(sAux[1]) do
            Faixa_FG.Add(Chr(i));
        end;
    6,7 : //Interpretar os operadores '+' e '?'
        begin
          Achou := False;
          i := Length(DefExpressao_FG);
          {Posiciona no último Caracter <> ''}
          while (not Achou) and (i > 0) do
            begin
              if DefExpressao_FG[i] = ' ' then
                Dec(i)
              else
                Achou := True;
            end;

          Achou := False;
          {Se existir parenteses}
          if DefExpressao_FG[i] = ')' then
            begin
              PAbre  := 0;
              PFecha := 1;
              while (not Achou) and (i > 0) do
                begin
                  Dec(i);
                  if DefExpressao_FG[i] = '(' then
                    inc(PAbre)
                  else
                    if DefExpressao_FG[i] = ')' then
                      inc(PFecha);
                  if PAbre = PFecha then
                    Achou := True;
                end;
            end
          {Se não existir parenteses}
          else
            while (not Achou) and (i > 0) do
              begin
                Dec(i);
                if (i = 0) or (DefExpressao_FG[i] in [' ',')','*','|','&','^']) then
                  begin
                    inc(i);
                    Achou := True;
                  end;
```

```
              end;

          {Agora que i = inicio da expressao que possui o operador '+' ou '?',
           será incluído o simb. '(' antes de i}
          Insert('(', DefExpressao_FG, i);
          if pTipo = 6 then
            begin
              sAux := Copy(DefExpressao_FG, i+1, Length(DefExpressao_FG)-i);
              DefExpressao_FG := DefExpressao_FG + '('+sAux+')*)';
            end
          else
            DefExpressao_FG := DefExpressao_FG + '|^)';
        end;
  end;
end;


(*******Consiste a existência do Ident. da Expressão Regular*******)
procedure T-->Grammar<--.ConsisteIdentER;
var
  Pos : Integer;
begin
  IdentExpressao := LexString;
  {Consiste existência}
  if vListaExpReg.Existe_ExpReg(LexString, '', tIdent, Pos) then
    begin
      InsereListaER_Dup(IdentExpressao, '');
      SynError(202);
    end;
end;

(*******Consiste se o Ident. utiliado na definição da ER, já foi declarado*******)
function T-->Grammar<--.ConsisteIdentER_NaDefinicao : Integer;
var
  Pos : Integer;
begin
  Result := 0;
  {Consiste existência}
  if not vListaExpReg.Existe_ExpReg(LexString, '', tIdent, Pos) then
    begin
      InsereListaER_Dup(LexString, '');
      SynError(208);
    end
  else
    begin
      Result := Pos;
      vListaExpReg.AlteraEh_ExpReg_Prim(Pos);
    end;
end;

(*******Consiste a existência e insere a Expressão Regular na lista*******)
procedure T-->Grammar<--.InsereListaER;
var
  vExpReg, vExpRegAux : TExpReg;
  Pos : Integer;
begin
  if not vListaExpReg.Existe_ExpReg(IdentExpressao, DefExpressao, tTodos, Pos) then
    begin
      vExpRegAux := TExpReg(vListaExpReg.Existe_ExpReg_Inicia(ListaSimbIniER));
      DefExpressao_FG := '(' + DefExpressao_FG + ')';
      vExpReg := TExpreg.CrieInicialize(IdentExpressao, DefExpressao, DefExpressao_FG,
ListaSimbIniER, SimbFimER, False);
      vListaExpReg.ColoqueElemento(vExpReg);

      if (vExpRegAux <> nil) and (not vExpRegAux.Eh_ExpReg_Prim) then
        begin
        InsereListaER_Dup(vExpRegAux.Ident_ExpReg, vExpRegAux.Def_ExpReg);
        InsereListaER_Dup(vExpRegAux.Ident_ExpReg, vExpRegAux.Def_ExpReg);
        InsereListaER_Dup(IdentExpressao, DefExpressao);
        InsereListaER_Dup(IdentExpressao, DefExpressao);
        SynError(210);
        end;
    end
  else
```

```
    begin
      InsereListaER_Dup(IdentExpressao, DefExpressao);
      SynError(203);
    end;
  IdentExpressao := '';
  DefExpressao := '';
  DefExpressao_FG := '';
  ListaSimbIniER.Clear;
end;

(*******Insere na Lista de Palavras Reservadas Duplicadas*******)
procedure T-->Grammar<--.InsereListaPR_Dup(pStrPRDup : String);
begin
  New(PString);
  PString^ := pStrPRDup;
  ListaPR_Dup.Add(PString);
end;

(*******Retina uma Palavra Reservada Duplicada da Lista*******)
function T-->Grammar<--.RetornaUltPR_Dup : String;
begin
  PString := ListaPR_Dup.Items[ListaPR_Dup.Count - 1];
  Result := PString^;
  Dispose(PString);
  ListaPR_Dup.Delete(ListaPR_Dup.Count - 1);
end;

(*******Insere na Lista de Simbolos Especiais Duplicados*******)
procedure T-->Grammar<--.InsereListaSE_Dup(pStrSEDup : String);
begin
  New(PString);
  PString^ := pStrSEDup;
  ListaSE_Dup.Add(PString);
end;

(*******Retina um Simbolos Especial Duplicado da Lista*******)
function T-->Grammar<--.RetornaUltSE_Dup : String;
begin
  PString := ListaSE_Dup.Items[ListaSE_Dup.Count - 1];
  Result := PString^;
  Dispose(PString);
  ListaSE_Dup.Delete(ListaSE_Dup.Count - 1);
end;

(*******Retina uma Expressão Regular Duplicada da Lista*******)
function T-->Grammar<--.RetornaUltER_Dup(IndiceCampo : Integer) : String;
begin
  RegER := ListaER_Dup.Items[0];
  case IndiceCampo of
   0 : Result := RegER^.IdentER;
   1 : Result := RegER^.DefER;
  end;
  Dispose(RegER);
  ListaER_Dup.Delete(0);
end;

(*******Insere na Lista de Caracteres Duplicadas*******)
procedure T-->Grammar<--.InsereListaCaracter(Caracter : Char);
begin
  New(PChar);
  PChar^ := Caracter;
  ListaCaracter.Add(PChar);
end;

(*******Retina um Caracter duplicado na faixa da Lista*******)
function T-->Grammar<--.RetornaCaracter : String;
begin
  PChar := ListaCaracter.Items[0];
  Result := Chr(39)+PChar^+Chr(39);
  Dispose(PChar);
  ListaCaracter.Delete(0);
end;

(*******Verifica se o caracter é o último antes do simb. final ">"*******)
```

```
procedure TMeta_Linguagem.EscreveUltimoCaracter;
var
  Aux, Pos : Integer;
  Ch : Char;
begin
  pos := SourceStream.Position;
  Aux := Pos;
  Ch := ' ';
  while Ch in [' ',#13,#10] do
    begin
    SourceStream.Seek(pos, soFromBeginning);
    SourceStream.ReadBuffer(Ch, 1);
    inc(Pos);
    end;

  if Ch = '>' then
    SimbFimER := LexString[2];
  SourceStream.Position := Aux;
end;

(*******Consiste se a faixa de caracter ou o caracter não foi informada*******)
procedure T-->Grammar<--.ConsisteCaracterFaixa(OrdemCaracter : Integer);
var
  i : Byte;
  Achou : Boolean;
begin
  if OrdemCaracter = 1 then
    begin
      Caracter1 := LexString[2];
      if Ord(Caracter1) in CnjCaraterASCII then
        begin
        AchouCaracter1 := True;
        InsereListaCaracter(Caracter1);
        SynError(204);
        end
      else
        begin
        AchouCaracter1 := False;
        CnjCaraterASCII := CnjCaraterASCII + [Ord(Caracter1)];
        end;
    end
  else
    if AchouCaracter1 then
      AchouCaracter1 := False
    else
      begin
        Caracter2 := LexString[2];
        if Ord(Caracter1) >= ord(Caracter2) then
          begin
            InsereListaCaracter(Caracter1);
            InsereListaCaracter(Caracter2);
            if Ord(Caracter1) > ord(Caracter2) then
              SynError(205)
            else
              SynError(206);
          end;
        Achou := False;
        i := ord(Caracter1) + 1;
        while (not Achou) and (i <= ord(Caracter2)) do
          if i in CnjCaraterASCII then
            Achou := True
          else
            begin
              CnjCaraterASCII := CnjCaraterASCII + [i];
              inc(i);
            end;
        if Achou then
          begin
            InsereListaCaracter(chr(i));
            SynError(207);
          end;
      end;
end;
```

```
(* Fim  - Código Form *)

(* Inicio - Especificação Léxica *)

IGNORE CASE

CHARACTERS
  Maius   = CHR(65)..CHR(90) .
  Minus   = CHR(97)..CHR(122) .
  Digit   = CHR(48)..CHR(57) .
  Aspa    = CHR(39).
  Aspas   = CHR(34).
  EOL     = CHR(13).
  N1      = CHR(38).  // &
  N2      = CHR(40).  // (
  N3      = CHR(41).  // )
  N4      = CHR(42).  // *
  N5      = CHR(94).  // ^
  N6      = CHR(124). // |
  ASCII   = ANY - EOL.
  Def_Lit = ASCII - Aspas - N1 - N2 - N3 - N4 - N5 - N6.
  Def_Car = ASCII - N1 - N2 - N3 - N4 - N5 - N6.


TOKENS
  Ident     = Maius | Minus { "_" | Maius | Minus | Digit}.
  Literal   = Aspas Def_Lit {Def_Lit} Aspas .
  Caracter = Aspa Def_Car Aspa.

COMMENTS FROM "//" TO EOL
COMMENTS FROM "/*" TO "*/" NESTED

IGNORE CHR(1)..CHR(32)

(* Fim - Especificação Léxica *)

(* Inicio - Especificação Sintática *)


PRODUCTIONS

 Meta_Linguagem =                      (. Inicializar; .)
                  "PALAVRAS"
                  "RESERVADAS"
                  ":"
                  { "("
                    "CAIXA"
                    "SENSIVEL"
                    ")"                (. CaseSensitive := True; .)
                  }
                  "{"                  (. TipoLiteral := tPR; .)
                  [Cnj_Literal]
                  "}"
                  "SIMBOLOS"
                  "ESPECIAIS"
                  ":"
                  "{"                  (. TipoLiteral := tSE; .)
                  [Cnj_Literal]
                  "}"
                  "TOKENS"
                  ":"
                  "{"
                  Cnj_Token
                  "}"
                                       (. Finalizar; .)
                  .
  Cnj_Literal   = Literal              (. InsereListaLiteral; .)
                  { ","
                    Literal            (. InsereListaLiteral; .)
                  }
                  .
  Cnj_Token     = "<"
                  Ident                (. ConsisteIdentER; .)
                  ":"                  (. Nivel := 1;
```

```
                                        SimbFimER := #32; .)
                  Expr_Reg              (. InsereListaER; .)
                  ">"
                  { "<"
                    Ident               (. ConsisteIdentER; .)
                    ":"                 (. Nivel := 1;
                                           SimbFimER := #32; .)
                    Expr_Reg            (. InsereListaER; .)
                    ">"
                  }
                  .
  Expr_Reg      = "("                   (. SomaER(1);
                                           inc(Nivel); .)
                  Expr_Reg
                  ")"                   (. SomaER(1);
                                           inc(Nivel); .)
                  [ ER_Linha ]
                | ER_Final              (. Inc(Nivel); .)
                  [ ER_Linha
                  ]
                  .
  ER_Linha      = Oper_Pos [ ER_Linha ]
                | ["|"                  (. SomaER(1);
                                           Dec(Nivel); .)
                  ]
                  Expr_Reg
                  .
  ER_Final      = Faixa
                | Ident                 (. PosER := ConsisteIdentER_NaDefinicao;
                                           SomaER(3); .)
                | Literal               (. SomaER(1);
                                           if Nivel = 1 then
                                             ListaSimbIniER.Add(LexString[2]); .)
                | Caracter              (. SomaER(2);
                                           if Nivel = 1 then
                                             ListaSimbIniER.Add(LexString[2]);
                                           EscreveUltimoCaracter; .)
                  .
  Faixa         =                       (. CnjCaraterASCII := [];
                                           AchouCaracter1 := False;
                                           OperNeg := False;
                                           Faixa_FG.Clear; .)
                  ["~"                  (. SomaER(0);
                                           OperNeg := True; .)
                  ]
                  "["                   (. SomaER(0); .)
                  Sub_Faixa
                  { ","                 (. SomaER(0); .)
                    Sub_Faixa}
                  "]"                   (. SomaER(0);
                                           MontaFaixa_FG; .)
                  .
  Sub_Faixa     = Caracter              (. SomaER(4);
                                           ConsisteCaracterFaixa(1); .)
                  ["."                  (. SomaER(0);.)
                   "."                  (. SomaER(0); .)
                   Caracter             (. SomaER(5);
                                           ConsisteCaracterFaixa(2); .)
                  ]
                  .
  Oper_Pos      = "+"                   (. SomaER(6);.)
                | "*"                   (. SomaER(1);.)
                | "?"                   (. SomaER(7);.)
                  .

END Meta_Linguagem.

(* Fim - Especificação Sintática *)
```

APÊNDICE 2 – CÓDIGO FONTE DA CLASSE GERADA *TANALISALEX*

Neste anexo é mostrada a classe gerada a partir da especificação apresentada na fig. 3.4.

```
Unit UntAnalisaLex;

interface

uses Classes, Dialogs, Sysutils, UntArrayTokens, UntToken;

Type

  TAnalisaLex = Class(TPersistent)
  Private
    FListaPR : TStringList;
    FListaSE : TStringList;
    FListaIdER : TStringList;
    FCnjListaSimbIniER : TList;
    FListaFimER : TStringList;
    FCaseSensitive : Boolean;
    FArrayTokens  : TArrayTokens;
    FTexto   : String;
    Procedure InsereTokenLista(pCadeia : String; pLin : Integer);
    function ValidaToken(Cadeia: String): Boolean;
  published
    constructor CrieInstancia;
    destructor Destrue_se;
    property Texto : String read FTexto write FTexto;
    property ArrayTokens : TArrayTokens read FArrayTokens write FArrayTokens;
    procedure MonteArrayTokens;
  end;


implementation

{ TAnalisaLex }

constructor TAnalisaLex.CrieInstancia;
var
  vListaSimbIniER : TStringList;
begin
  inherited;
  FListaPR := TStringList.Create;
  FListaSE := TStringList.Create;
  FListaIdER := TStringList.Create;
  FCnjListaSimbIniER := TList.Create;
  FListaFimER := TStringList.Create;

  FListaPR.Add('program');
  FListaPR.Add('var');
  FListaSE.Add('{');
  FListaSE.Add(':=');
  FListaSE.Add(';');
  FListaSE.Add('}');
  FListaIdER.Add('identificador');
  vListaSimbIniER := TStringList.Create;
  vListaSimbIniER.Add('a');
  vListaSimbIniER.Add('b');
  vListaSimbIniER.Add('c');
  vListaSimbIniER.Add('d');
```

```
  vListaSimbIniER.Add('e');
  vListaSimbIniER.Add('f');
  vListaSimbIniER.Add('g');
  vListaSimbIniER.Add('h');
  vListaSimbIniER.Add('i');
  vListaSimbIniER.Add('j');
  vListaSimbIniER.Add('k');
  vListaSimbIniER.Add('l');
  vListaSimbIniER.Add('m');
  vListaSimbIniER.Add('n');
  vListaSimbIniER.Add('o');
  vListaSimbIniER.Add('p');
  vListaSimbIniER.Add('q');
  vListaSimbIniER.Add('r');
  vListaSimbIniER.Add('s');
  vListaSimbIniER.Add('t');
  vListaSimbIniER.Add('u');
  vListaSimbIniER.Add('v');
  vListaSimbIniER.Add('w');
  vListaSimbIniER.Add('x');
  vListaSimbIniER.Add('y');
  vListaSimbIniER.Add('z');
  vListaSimbIniER.Add('A');
  vListaSimbIniER.Add('B');
  vListaSimbIniER.Add('C');
  vListaSimbIniER.Add('D');
  vListaSimbIniER.Add('E');
  vListaSimbIniER.Add('F');
  vListaSimbIniER.Add('G');
  vListaSimbIniER.Add('H');
  vListaSimbIniER.Add('I');
  vListaSimbIniER.Add('J');
  vListaSimbIniER.Add('K');
  vListaSimbIniER.Add('L');
  vListaSimbIniER.Add('M');
  vListaSimbIniER.Add('N');
  vListaSimbIniER.Add('O');
  vListaSimbIniER.Add('P');
  vListaSimbIniER.Add('Q');
  vListaSimbIniER.Add('R');
  vListaSimbIniER.Add('S');
  vListaSimbIniER.Add('T');
  vListaSimbIniER.Add('U');
  vListaSimbIniER.Add('V');
  vListaSimbIniER.Add('W');
  vListaSimbIniER.Add('X');
  vListaSimbIniER.Add('Y');
  vListaSimbIniER.Add('Z');
  FCnjListaSimbIniER.Add(vListaSimbIniER);
  FListaFimER.Add(' ');
  FListaIdER.Add('inteiro');
  vListaSimbIniER := TStringList.Create;
  vListaSimbIniER.Add('0');
  vListaSimbIniER.Add('1');
  vListaSimbIniER.Add('2');
  vListaSimbIniER.Add('3');
  vListaSimbIniER.Add('4');
  vListaSimbIniER.Add('5');
  vListaSimbIniER.Add('6');
  vListaSimbIniER.Add('7');
  vListaSimbIniER.Add('8');
  vListaSimbIniER.Add('9');
  FCnjListaSimbIniER.Add(vListaSimbIniER);
  FListaFimER.Add(' ');
end;

destructor TAnalisaLex.Destrue_se;
begin
  inherited;
  FListaPR.Free;
  FListaSE.Free;
  FListaIdER.Free;
  FCnjListaSimbIniER.Free;
  FListaFimER.Free;
```

```
  FArrayTokens.Destrua_se;
end;

procedure TAnalisaLex.MonteArrayTokens;

  procedure InicializaArrayTokens;
  begin
    if FArrayTokens = nil then
      FArrayTokens := TArrayTokens.CrieInstancia
    else
      begin
      FArrayTokens.Destrua_se;
      FArrayTokens := TArrayTokens.CrieInstancia
      end;
  end;

  function ProcuraSE(C, C2 : Char) : Integer;
  var
    j : LongInt;
  begin
    j := 0;
    Result := -1;
    while (j < FListaSE.Count) and (Result < 0) do
      begin
      if (C = FListaSE[j][1]) then
        if C2 = ' ' then
          Result := j
        else
          if (Length(FListaSE[j]) = 2) and (C2 = FListaSE[j][2]) then
            Result := j;
      inc(j);
      end;
  end;

  procedure ExisteER_Com_IniFim(C  : Char; var SimbFim : Char);
  var
    j : LongInt;
    vLista : TStringList;
  begin
    SimbFim := ' ';
    j := 0;
    vLista := TStringList.Create;
    try
    while (j < FCnjListaSimbIniER.Count) and (SimbFim = ' ') do
      begin
      {Se contem um único caracter que pode iniciar}
      if TStringList(FCnjListaSimbIniER[j]).Count = 1 then
        {Verifica se é igual ao caracter lido}
        if C = TStringList(FCnjListaSimbIniER[j])[0] then
          SimbFim := TStringList(FCnjListaSimbIniER[j])[0][1];
      inc(j);
      end;
    finally
      vLista.Free;
    end;
  end;

var
  Cadeia : String;
  C, C2 : Char;
  PosAchou, PosAchou2, Lin : Integer;
  Tam, i, j : LongInt;
  vToken : TToken;
  SimbFim : Char;
begin
  InicializaArrayTokens;
  FTexto := Trim(FTexto);
  vToken := TToken.Create;
  Lin := 1;
  i := 1;
  Cadeia := '';
  Tam := Length(FTexto);
  SimbFim := ' ';
  while i <= Tam do
```

```
  begin
    C := FTexto[i];
    PosAchou := ProcuraSE(C, ' ');
    if (PosAchou >= 0)and(SimbFim = ' ') then
      begin
      C2 := FTexto[i+1];
      PosAchou2 := ProcuraSE(C, C2);
      if PosAchou2 >= 0 then
        begin
        InsereTokenLista(C+C2, Lin);
        inc(i);
        end
      else
        InsereTokenLista(C, Lin);
      end
    else
      if not(C in [#13, #10, ' ']) then
        begin
        if Cadeia = '' then
          begin
          ExisteER_Com_IniFim(C, SimbFim);
          Cadeia := Cadeia + C;
          end
        else
          begin
          Cadeia := Cadeia + C;
          if C = SimbFim then
            begin
            InsereTokenLista(Cadeia, Lin);
            Cadeia := '';
            SimbFim := ' ';
            end;
          end;

        if Cadeia <> '' then
        {Se estiver lendo o último caracter}
          if i+1 > Tam then
            InsereTokenLista(Cadeia, Lin)
          else
            {Se não for o último}
            begin
            C2 := FTexto[i+1];
            PosAchou2 := ProcuraSE(C2, ' ');
            {Se (prox simb é um SE e a cadeia não possui simb final) ou (prox Simb é nulo}
            if ((PosAchou2 >= 0)and(SimbFim = ' ')) or (C2 in [#13, #10, ' ']) then
              begin
              InsereTokenLista(Cadeia, Lin);
              Cadeia := '';
              end;
            end;
        end
      else
        if C2 = #10 then
          inc(Lin);
    inc(i);
  end;
end;

procedure TAnalisaLex.InsereTokenLista(pCadeia : String; pLin : Integer);

  function RetornaFListaIdER(pCadeia : String) : String;
  var
    j, i : Integer;
    vLista : TStringList;
  begin
    Result := '';
    j := 0;
    vLista := TStringList.Create;
    try
      while (j < FListaIdER.Count) and (Result = '') do
        begin
        i := 0;
        vLista := FCnjListaSimbIniER[j];
        while (i < vLista.Count) and (Result = '') do
```

```
          begin
          if pCadeia[1] = vLista[i] then
            Result := FListaIdER[j];
          inc(i);
          end;
        inc(j);
        end;
      if Result = '' then
        Result := FListaIdER[0];
    finally
    end;
  end;

var
  Pos : Integer;
  Token : TToken;
  vIdentER : String;
begin
  {Cria Token}
  Token := TToken.Create;
  Token.Ident := pCadeia;
  Token.Linha := pLin;

  //Verifica se a cadeia é uma SE
  Pos := FListaSE.IndexOf(pCadeia);
  if Pos >= 0 then
    begin
    Token.NumCateg := FListaPR.Count + Pos + 1;
    Token.DesCateg := FListaSE[Pos];
    end
  else
    begin
    {Verifica se a cadeia é uma PR}
    if FCaseSensitive then
      Pos := FListaPR.IndexOf(pCadeia)
    else
      Pos := FListaPR.IndexOf(UpperCase(pCadeia));
    if Pos >= 0 then
      begin
      Token.NumCateg := Pos + 1;
      Token.DesCateg := FListaPR[Pos];
      end
    else
      begin
      //Verifica se a cadeia é um Token
      Token.DesCateg := RetornaFListaIdER(pCadeia);
      Pos := FListaIdER.IndexOf(Token.DesCateg);
      if ValidaToken(pCadeia) then
        Token.NumCateg := FListaPR.Count + FListaSE.Count + Pos + 1
      else
        Token.NumCateg := 0;
      end;
    end;
  {Insere Token na Lista}
  FArrayTokens.Add_Token(Token)
end;


function TAnalisaLex.ValidaToken(Cadeia: String): Boolean;
{ Estados do AFD }
label S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10;
var
  I      : integer;
begin
  S0: I := 0;
  S1: inc(I);
      if length(Cadeia) < I then
        goto S9;
      if Cadeia[I] = 'a' then
        goto S2;
      if Cadeia[I] = 'b' then
        goto S2;
      if Cadeia[I] = 'c' then
        goto S2;
```

```
      if Cadeia[I] = 'd' then
        goto S2;
      if Cadeia[I] = 'e' then
        goto S2;
      if Cadeia[I] = 'f' then
        goto S2;
      if Cadeia[I] = 'g' then
        goto S2;
      if Cadeia[I] = 'h' then
        goto S2;
      if Cadeia[I] = 'i' then
        goto S2;
      if Cadeia[I] = 'j' then
        goto S2;
      if Cadeia[I] = 'k' then
        goto S2;
      if Cadeia[I] = 'l' then
        goto S2;
      if Cadeia[I] = 'm' then
        goto S2;
      if Cadeia[I] = 'n' then
        goto S2;
      if Cadeia[I] = 'o' then
        goto S2;
      if Cadeia[I] = 'p' then
        goto S2;
      if Cadeia[I] = 'q' then
        goto S2;
      if Cadeia[I] = 'r' then
        goto S2;
      if Cadeia[I] = 's' then
        goto S2;
      if Cadeia[I] = 't' then
        goto S2;
      if Cadeia[I] = 'u' then
        goto S2;
      if Cadeia[I] = 'v' then
        goto S2;
      if Cadeia[I] = 'w' then
        goto S2;
      if Cadeia[I] = 'x' then
        goto S2;
      if Cadeia[I] = 'y' then
        goto S2;
      if Cadeia[I] = 'z' then
        goto S2;
      if Cadeia[I] = '0' then
        goto S3;
      if Cadeia[I] = '1' then
        goto S3;
      if Cadeia[I] = '2' then
        goto S3;
      if Cadeia[I] = '3' then
        goto S3;
      if Cadeia[I] = '4' then
        goto S3;
      if Cadeia[I] = '5' then
        goto S3;
      if Cadeia[I] = '6' then
        goto S3;
      if Cadeia[I] = '7' then
        goto S3;
      if Cadeia[I] = '8' then
        goto S3;
      if Cadeia[I] = '9' then
        goto S3;
      goto S9;
S2:   inc(I);
      if length(Cadeia) < I then
        goto S8;
      if Cadeia[I] = 'a' then
        goto S4;
      if Cadeia[I] = 'b' then
        goto S4;
```

```
      if Cadeia[I] = 'c' then
        goto S4;
      if Cadeia[I] = 'd' then
        goto S4;
      if Cadeia[I] = 'e' then
        goto S4;
      if Cadeia[I] = 'f' then
        goto S4;
      if Cadeia[I] = 'g' then
        goto S4;
      if Cadeia[I] = 'h' then
        goto S4;
      if Cadeia[I] = 'i' then
        goto S4;
      if Cadeia[I] = 'j' then
        goto S4;
      if Cadeia[I] = 'k' then
        goto S4;
      if Cadeia[I] = 'l' then
        goto S4;
      if Cadeia[I] = 'm' then
        goto S4;
      if Cadeia[I] = 'n' then
        goto S4;
      if Cadeia[I] = 'o' then
        goto S4;
      if Cadeia[I] = 'p' then
        goto S4;
      if Cadeia[I] = 'q' then
        goto S4;
      if Cadeia[I] = 'r' then
        goto S4;
      if Cadeia[I] = 's' then
        goto S4;
      if Cadeia[I] = 't' then
        goto S4;
      if Cadeia[I] = 'u' then
        goto S4;
      if Cadeia[I] = 'v' then
        goto S4;
      if Cadeia[I] = 'w' then
        goto S4;
      if Cadeia[I] = 'x' then
        goto S4;
      if Cadeia[I] = 'y' then
        goto S4;
      if Cadeia[I] = 'z' then
        goto S4;
      if Cadeia[I] = '0' then
        goto S4;
      if Cadeia[I] = '1' then
        goto S4;
      if Cadeia[I] = '2' then
        goto S4;
      if Cadeia[I] = '3' then
        goto S4;
      if Cadeia[I] = '4' then
        goto S4;
      if Cadeia[I] = '5' then
        goto S4;
      if Cadeia[I] = '6' then
        goto S4;
      if Cadeia[I] = '7' then
        goto S4;
      if Cadeia[I] = '8' then
        goto S4;
      if Cadeia[I] = '9' then
        goto S4;
      if Cadeia[I] = '_' then
        goto S4;
      goto S9;
  S3: inc(I);
      if length(Cadeia) < I then
        goto S8;
```

```
      if Cadeia[I] = '0' then
        goto S5;
      if Cadeia[I] = '1' then
        goto S5;
      if Cadeia[I] = '2' then
        goto S5;
      if Cadeia[I] = '3' then
        goto S5;
      if Cadeia[I] = '4' then
        goto S5;
      if Cadeia[I] = '5' then
        goto S5;
      if Cadeia[I] = '6' then
        goto S5;
      if Cadeia[I] = '7' then
        goto S5;
      if Cadeia[I] = '8' then
        goto S5;
      if Cadeia[I] = '9' then
        goto S5;
      goto S9;
  S4: inc(I);
      if length(Cadeia) < I then
        goto S8;
      if Cadeia[I] = 'a' then
        goto S4;
      if Cadeia[I] = 'b' then
        goto S4;
      if Cadeia[I] = 'c' then
        goto S4;
      if Cadeia[I] = 'd' then
        goto S4;
      if Cadeia[I] = 'e' then
        goto S4;
      if Cadeia[I] = 'f' then
        goto S4;
      if Cadeia[I] = 'g' then
        goto S4;
      if Cadeia[I] = 'h' then
        goto S4;
      if Cadeia[I] = 'i' then
        goto S4;
      if Cadeia[I] = 'j' then
        goto S4;
      if Cadeia[I] = 'k' then
        goto S4;
      if Cadeia[I] = 'l' then
        goto S4;
      if Cadeia[I] = 'm' then
        goto S4;
      if Cadeia[I] = 'n' then
        goto S4;
      if Cadeia[I] = 'o' then
        goto S4;
      if Cadeia[I] = 'p' then
        goto S4;
      if Cadeia[I] = 'q' then
        goto S4;
      if Cadeia[I] = 'r' then
        goto S4;
      if Cadeia[I] = 's' then
        goto S4;
      if Cadeia[I] = 't' then
        goto S4;
      if Cadeia[I] = 'u' then
        goto S4;
      if Cadeia[I] = 'v' then
        goto S4;
      if Cadeia[I] = 'w' then
        goto S4;
      if Cadeia[I] = 'x' then
        goto S4;
      if Cadeia[I] = 'y' then
        goto S4;
```

```
      if Cadeia[I] = 'z' then
        goto S4;
      if Cadeia[I] = '0' then
        goto S4;
      if Cadeia[I] = '1' then
        goto S4;
      if Cadeia[I] = '2' then
        goto S4;
      if Cadeia[I] = '3' then
        goto S4;
      if Cadeia[I] = '4' then
        goto S4;
      if Cadeia[I] = '5' then
        goto S4;
      if Cadeia[I] = '6' then
        goto S4;
      if Cadeia[I] = '7' then
        goto S4;
      if Cadeia[I] = '8' then
        goto S4;
      if Cadeia[I] = '9' then
        goto S4;
      if Cadeia[I] = '_' then
        goto S4;
      goto S9;
  S5: inc(I);
      if length(Cadeia) < I then
        goto S8;
      if Cadeia[I] = '0' then
        goto S6;
      if Cadeia[I] = '1' then
        goto S6;
      if Cadeia[I] = '2' then
        goto S6;
      if Cadeia[I] = '3' then
        goto S6;
      if Cadeia[I] = '4' then
        goto S6;
      if Cadeia[I] = '5' then
        goto S6;
      if Cadeia[I] = '6' then
        goto S6;
      if Cadeia[I] = '7' then
        goto S6;
      if Cadeia[I] = '8' then
        goto S6;
      if Cadeia[I] = '9' then
        goto S6;
      goto S9;
  S6: inc(I);
      if length(Cadeia) < I then
        goto S8;
      if Cadeia[I] = '0' then
        goto S7;
      if Cadeia[I] = '1' then
        goto S7;
      if Cadeia[I] = '2' then
        goto S7;
      if Cadeia[I] = '3' then
        goto S7;
      if Cadeia[I] = '4' then
        goto S7;
      if Cadeia[I] = '5' then
        goto S7;
      if Cadeia[I] = '6' then
        goto S7;
      if Cadeia[I] = '7' then
        goto S7;
      if Cadeia[I] = '8' then
        goto S7;
      if Cadeia[I] = '9' then
        goto S7;
      goto S9;
  S7: inc(I);
```

```
      if length(Cadeia) < I then
        goto S8;
      goto S9;
  S8: Result := true;
      goto S10;
  S9: Result := false;
  S10: ;
end;

end.
```

REFERÊNCIAS BIBLIOGRÁFICAS

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

CANTÙ, Marco. **Dominando o Delphi 5**. Tradução João Eduardo Nóbrega Tortello. São Paulo: Makron Books do Brasil, 2000.

COCOLSOFT COMPUTER SOLUTIONS. **Cogencee**. [2001]. Disponível em: <http://www.cocolsoft.com.au/cogen/cogen.htm>. Acesso em: jul. 2002.

FURLAN, José D. **Modelagem de objetos através da UML:** the unified modeling language. São Paulo: Makron Books, 1998.

GERSTING, Judith L. **Fundamentos matemáticos para ciência da computação.** 3. Ed. Rio de Janeiro : LTC, 1995.

GLATZ, Ronald. **Protótipo para transformação de uma expressão regular para uma função equivalente em Pascal, utilizando dois algoritmos baseados no teorema de Kleene**. 2000. 104 f. Trabalho de Conclusão de Curso (Bacharelado em Ciências da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

GRUNE, Dick. **Projeto moderno de compiladores:** implementação e aplicações. Tradução Vanderberg D. de Souza. Rio de Janeiro: Campus, 2001.

HOPCROFT, J. E.; ULLMANN, J.D. **Introduction to automato theory, languages and computation.** Massachusets : Addison Wesley, 1979.

LEWIS, Harry R.; PAPADIMITRIOU, Christos H. **Elementos de teoria da computação.** 2. ed. São Paulo: Bookman, 2000.

MANNA, Zohar. **Mathematical theory of computation.** New York : McGraw-Hill, 1974.

MARTINS, Joyce. **Linguagens formais e compiladores**. [2002]. 43 f. Notas de Aula (Curso de Ciências da Computação) – Universidade Regional de Blumenau, Blumenau, SC. Disponível em: <http://www.inf.furb.br/~joyce>. Acesso em: jun. 2002.

MENEZES, Paulo B. **Linguagens formais e autômatos.** Porto Alegre: Sagra-Luzzatto, 1998.

MÖSSENBÖCK, Hanspeter et al. **Coco/R**. [2002]. Disponível em: <http://cs.ru.ac.za/homes/cspt/cocor.htm>. Acesso em: jul. 2002.

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.

REITH, Michael W. **TetZel Inscriptions** – CocoR for Delphi. [2002]. Disponível em: <http://www.tetzel.com/CocoR/>. Acesso em: out. 2002.

SILVA, José R.V. **Proposta de um novo algoritmo para transformação de uma expressão regular em um autômato finito determinístico**. Artigo não publicado. Blumenau: Universidade Regional de Blumenau, 2000.

THIRY, Marcello; SALM JR., José. **Processo de desenvolvimento de software com UML**. [2001]. Disponível em: < http://www.eps.ufsc.br/disc/procuml/>. Acesso em: nov.2002.

WEBGAIN INC. **Java Compiler Compiler™ (JavaCC)** - the Java Parser Generator. [2002?]. Disponível em: <http://www.webgain.com/products/java_cc/>. Acesso em: jul. 2002.

ZUMER, Viljem et al. **LISA**: a tool for automatic language implementation. [?]. Disponível em: <http://marcel.uni-mb.si/nikolaj/sigplan.htm>. Acesso em: jul. 2002.