UNIVERSIDADE REGIONAL DE BLUMENAU

CENTRO DE CIÊNCIAS EXATAS E NATURAIS

CURSO DE CIÊNCIAS DA COMPUTAÇÃO – BACHARELADO

FERRAMENTA PARA DESENVOLVIMENTO DE OBJETOS COMPARTILHÁVEIS DE BANCO DE DADOS EM LINGUAGEM PROCEDURAL

DAVI RODRIGO BIANCHI

BLUMENAU
2006

2006/2-07

DAVI RODRIGO BIANCHI

FERRAMENTA PARA DESENVOLVIMENTO DE OBJETOS COMPARTILHÁVEIS DE BANCO DE DADOS EM LINGUAGEM PROCEDURAL

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

Prof. Alexander Roberto Valdameri - Orientador

BLUMENAU
2006

2006/2-07

FERRAMENTA PARA DESENVOLVIMENTO DE OBJETOS COMPARTILHÁVEIS DE BANCO DE DADOS EM LINGUAGEM PROCEDURAL

Por

DAVI RODRIGO BIANCHI

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

Presidente:	Prof. Alexander Roberto Valdameri , Mestre – Orientador, FURB
Membro:	Prof. Maurício Capobianco Lopes, Mestre – FURB
Membro:	Prof. Wilson Pedro Carli, Mestre – FURB

Blumenau, 29/11/2006

Dedico este trabalho aos familiares, amigos, e a todos aqueles que de alguma forma deram seu apoio e compreensão durante o período de graduação.

AGRADECIMENTOS

À minha família, que apesar da distância, sempre esteve ao meu lado presente e dando apoio durante todos esses anos.

À minha namorada, Renata, sempre carinhosa e compreensiva.

Aos meus amigos, pelo companheirismo e ajuda prestada em todo período acadêmico.

Aos professores do curso de Bacharelado em Ciências da Computação, participantes ativos em minha formação profissional.

Ao meu orientador e amigo, Alexander Roberto Valdameri, por acreditar na minha capacidade e me auxiliar sempre que necessário no desenvolvimento deste trabalho.

“O que você fará hoje lhe custará um dia de sua vida. É muito caro. Faça coisas que tenham valor.”

Tadeu Comerlatto

RESUMO

Este trabalho apresenta uma ferramenta que possibilita a escrita de um procedimento armazenado de banco de dados em uma linguagem padrão, oferecendo a possibilidade de geração de código para os SGBDs Oracle, MS SQL Server e PostgreSQL e a realização de conexão aos mesmos para criação dos procedimentos. Para desenvolvimento dessa ferramenta, fez-se uso da linguagem procedural para desenvolvimento objetos de banco de dados proposta por Hiebert (2003). Tal linguagem procedural sofreu incrementos significativos, oferecendo dessa forma uma gama mais ampla de comandos.

Palavras-chave: Linguagem procedural. Compiladores. SGBD. Geração de código.

ABSTRACT

This paper presents a tool that makes possible write a database stored procedure in a standard language, offering the possibility of code generation for SGBDs Oracle, MS SQL Server and PostgreSQL and the connection accomplishment to the same ones for procedure creation. For development of this tool, was made use of the procedural language for database objects development proposed by Hiebert (2003). Such procedural language suffered significant increments, offering a wider range of commands in that way.

Keywords: Procedural language. Compilers. DBMS. Code generation.

LISTA DE ILUSTRAÇÕES

Figura 1 – Estrutura de um compilador segundo Price e Toscani 19
Figura 2 – Um AFN que aceita *(a | b)* abb* 21
Figura 3 – AFD que aceita *aa* | bb** 21
Quadro 1 – Exemplo de uma GLC G = (V, T, P, S) 22
Quadro 2 – Exemplo de uma GLC utilizando a notação BNF 24
Figura 4 – Visão conceitual da tradução dirigida por sintaxe 26
Quadro 3 – Algoritmo para definição de atributos sintetizados e herdados num grafo de dependência 27
Figura 5 – Árvore de derivação e árvore de sintaxe para a sentença 4*2+8 27
Quadro 4 – Evolução da padronização da linguagem SQL 29
Quadro 5 – Tipos de dados SQL e representação lógica 31
Quadro 6 – Exemplo de procedimento armazenado utilizando a linguagem definida pela GLC do Apêndice A 32
Quadro 7 – Caso de uso analisar código 36
Quadro 8 – Restrições do caso de uso analisar código 36
Quadro 9 – Cenários do caso de uso analisar código 37
Quadro 10 – Caso de uso gerar código 37
Quadro 11 – Restrições do caso de uso gerar código 37
Quadro 12 – Cenários do caso de uso gerar código 37
Quadro 13 – Caso de uso efetuar login 38
Quadro 14 – Restrições do caso de uso efetuar login 38
Quadro 15 – Cenários do caso de uso efetuar login 38
Quadro 16 – Caso de uso criar procedimento 39
Quadro 17 – Restrições do caso de uso criar procedimento 39
Quadro 18 – Cenários do caso de uso criar procedimento 39
Quadro 19 – Diagrama de classes 40
Quadro 20 – Diagrama de atividades da geração de código 42
Quadro 21 – Diagrama de seqüências da geração de código 43
Quadro 22 – Definições regulares 44
Quadro 23 – Reconhecimento de *tokens* 44
Quadro 24 – Derivação do comando IF-THEN na análise LL(1) 46

Quadro 25 – Tela principal do GALS 47
Quadro 26 – *Header* da classe TGenOracle 48
Quadro 27 – Ações semânticas 48
Quadro 28 – Método *Write* 49
Figura 11 – Tela principal 50
Figura 12 – Verificação de erros 51
Figura 13 – Conexão com banco de dados 51
Figura 14 – Criação do procedimento 52
Quadro 29 – Fonte gerado para o SGBD Oracle 52
Quadro 30 – Fonte gerado para o SGBD MS SQL Server 53
Quadro 31 – Fonte gerado para o SGBD PostgreSQL 53
Quadro 32 – Regras sintáticas da linguagem 66

LISTA DE SIGLAS

AFD – Autômato Finito Determinístico

AFN – Autômato Finito não Determinístico

ANSI – *American National Standards Institute*

BNF – *Backus-Naur Form*

DDL – *Data-Definition Language*

DML – *Data-Manipulation Language*

DQL – *Data-Query Language*

GLC – Gramática Livre de Contexto

IBM – *International Business Machines Corporation*

IEC – *International Engineering Consortium*

ISO – *International Standard Organization*

LL – *Left to Right with Leftmost Derivation*

PL/pgSQL – *Procedural Language/Postgre Structured Query Language*

PL/SQL – *Procedural Language/Structured Query Language*

SDO – *Shared Database Object*

SGBD – Sistema Gerenciador de Banco de Dados

SEQUEL – *Structured English Query Language*

SQL – *Structured Query Language*

UML – *Unified Modeling Language*

LISTA DE SÍMBOLOS

@ - arroba

; - ponto-vírgula

ε – vazio

SUMÁRIO

1 INTRODUÇÃO 14
1.1 OBJETIVOS DO TRABALHO 15
1.2 ESTRUTURA DO TRABALHO 16
1.3 MOTIVAÇÃO 16
2 FUNDAMENTAÇÃO TEÓRICA 18
2.1 COMPILADORES 18
2.1.1 As Fases de um Compilador 18
2.1.2 Análise Léxica 20
2.1.3 Autômatos Finitos 20
2.1.4 Análise Sintática 21
2.1.5 Gramáticas Livres de Contexto 22
2.1.6 Análise Descendente (*Top-down*) 24
2.1.7 Tradução Dirigida por Sintaxe 25
2.1.8 Árvores de Sintaxe 27
2.1.9 Geração de Código 28
2.2 LINGUAGENS DE BANCO DE DADOS 28
2.2.1 A Linguagem SQL 29
2.2.2 Sub-linguagens 30
2.2.3 Tipos de Dados 30
2.3 PROCEDIMENTOS ARMAZENADOS 31
2.4 TRABALHOS CORRELATOS 34
3 DESENVOLVIMENTO DO TRABALHO 35
3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO 35
3.2 ESPECIFICAÇÃO 36
3.2.1 Diagramas de caso de uso 36
3.2.2 Diagrama de classes 39
3.2.3 Diagrama de atividades 42
3.2.4 Diagrama de Seqüências 42
3.2.5 Autômato finito 44
3.2.6 Analisador sintático 45
3.3 IMPLEMENTAÇÃO 45

3.3.1 Técnicas e ferramentas utilizadas.....45
3.3.2 Operacionalidade da implementação 50
3.4 RESULTADOS E DISCUSSÃO 54
4 CONCLUSÕES.....55
4.1 EXTENSÕES 56
REFERÊNCIAS BIBLIOGRÁFICAS 57
APÊNDICE A – GRAMÁTICA DA LINGUAGEM DE BANCO DE DADOS.....59

1 INTRODUÇÃO

Os crescentes avanços tecnológicos e a evolução constante dos sistemas computacionais geraram a demanda de um local de armazenamento de dados seguro e capaz de manipular de forma ágil e rápida grandes quantidades de informação. Para sanar tal necessidade, a utilização de um Sistema Gerenciador de Banco de Dados (SGDB) torna-se imprescindível.

> A razão de ser de um sistema gerenciador de banco de dados é gerenciar os dados armazenados em um banco de dados. Em geral, pode ser visto sob duas perspectivas: a do usuário e a do SGBD. Os usuários visualizam um banco de dados como uma coleção de dados logicamente agrupados. Para um SGBD, um banco de dados é simplesmente uma série de bytes, usualmente armazenados em um disco rígido. (PETKOVIC, 2001, p. 5).

Através da possibilidade de criação de estruturas de controle internas, os SGBDs passaram a suportar parte do desenvolvimento da aplicação. Além disso, como aspectos relevantes na utilização de um SGDB, pode-se citar, por exemplo:

a) consistência de dados;
b) segurança e autorização;
c) integridade de dados;
d) independência física e lógica de dados;
e) controle concorrente.

Em Taylor (2003, p. 21) é afirmado que a *Structured Query Language* (SQL) é a ferramenta mais utilizada para comunicação em bancos de dados relacionais. De acordo com Kriegel e Trukhnov (2003, p. 42), a linguagem SQL foi projetada para armazenamento de dados, recuperação e manipulação, e como tal, foi incorporada aos SGBDs, ou seja, não existe fora de um SGBD, nem poderia ser executada sem.

Para escrever objetos compartilháveis de banco de dados existe uma linguagem, que estende a forma procedural do SQL, denominada pela Oracle como *Procedural Language/Structured Query Language* (PL/SQL). Conforme descrito em Oliveira (2000), a linguagem PL/SQL combina o poder de manipulação de dados do SQL com os recursos de uma linguagem de programação de alto nível.

No entanto, cada SGBD possui sintaxe própria para utilização de linguagem procedural, o que vai contra a padronização existente na linguagem SQL. Logo, o desenvolvimento dos objetos do banco de dados compartilháveis para diferentes SGBDs por parte do programador do sistema de informação torna-se uma tarefa árdua e complexa, uma

vez que as peculiaridades devem ser consideradas.

Visando criar um ambiente comum para programação de objetos de banco de dados, Hiebert (2003) aborda em seu trabalho de conclusão o desenvolvimento de um protótipo de compilador, cujo objetivo é a geração de código para dois SGBDs específicos, Oracle e MS SQL Server. A partir da definição da gramática de uma linguagem padrão e utilizando um código fonte obedecendo às regras dessa gramática, o protótipo realiza a verificação de erros e a geração de código para o SGBD específico, considerando as características do mesmo.

Para construção de tal ferramenta, Hiebert (2003) utilizou técnicas de compilação para validação do código fonte, gerando o respectivo código objeto para SGBDs distintos. Grune et al. (2001, p. 1) define compilador como “um programa que aceita como entrada um texto de programa em uma certa linguagem e produz como saída um texto de programa em outra linguagem, enquanto preserva o significado deste texto”.

Conforme Aho, Sethi e Ullman (1995, p. 3-5), conceitualmente, um compilador opera em fases, cada uma das quais transforma o programa fonte de uma representação para a outra. Na prática, algumas das fases podem ser agrupadas e a representação intermediária entre as mesmas não precisa ser explicitamente construída. Como parte do núcleo de análise do compilador, responsável pela validação do código fonte e detecção de erros, tem-se as fases de análise léxica e sintática. Ainda tem-se como fases não menos importantes, a análise semântica, a geração de código intermediário, a otimização de código e a fase de geração de código, que realizam a síntese do código fonte. Todas as fases podem encontrar erros, porém as fases de análise sintática e semântica tratam usualmente de uma ampla fatia de erros detectáveis pelo compilador.

1.1 OBJETIVOS DO TRABALHO

O objetivo deste trabalho é estender a gramática da linguagem procedural definida por Hiebert (2003, p. 49-52), assim como desenvolver um compilador para geração de código.

Os objetivos específicos do trabalho são:

a) compilar o código fonte, através de análise léxica e sintática, como forma de incrementar a detecção de erros;
b) geração de código objeto, suportando as novas implementações, para os SGBDs Oracle, MS SQL Server e PostgreSQL.

c) possibilitar a conexão aos SGBDs, permitindo a execução nos mesmos do código objeto gerado, como forma de validação.

1.2 ESTRUTURA DO TRABALHO

A composição do trabalho dá-se através de quatro capítulos, sendo que esse primeiro, apresenta a origem do trabalho, contextualização do tema proposto, os objetivos, a organização e a motivação.

No segundo capítulo são apresentados conceitos sobre compiladores, funcionamento de um compilador, algumas técnicas que podem ser utilizadas na construção de um compilador, características da linguagem de consulta SQL e conceitos sobre procedimentos armazenados de bancos de dados.

No terceiro capítulo é descrita a especificação da ferramenta e detalhes sobre a implementação e funcionamento da mesma.

Por fim, no quarto capítulo são apresentadas conclusões, limitações da ferramenta e sugestões para continuidade deste trabalho.

1.3 MOTIVAÇÃO

A indústria de software busca uma solução cada vez mais eficaz para atender a demanda dos clientes, que normalmente possuem uma necessidade específica para adaptar o seu negócio à tecnologia da informação por meio de recursos computacionais. Logo, a construção de um sistema que oferece suporte à utilização de diferentes SGBDs tornou-se uma alternativa eficiente, uma vez que o sistema pode se encaixar de forma mais fácil à tecnologia já empregada pelo cliente. Outro aspecto de grande interesse da indústria de software é a possibilidade da utilização de um SGBD *open source*, sendo esse livre de licença, impactando profundamente no custo final do sistema de informação.

Visando uma redução de erros e um aumento na agilidade do desenvolvimento de um sistema compatível com SGBDs distintos, a ferramenta proposta no trabalho tem o intuito de auxiliar na construção dos objetos de banco de dados, possibilitando a escrita dos mesmos em

uma linguagem padrão, com geração para a linguagem definida por cada um dos respectivos SGBDs, acelerando o processo de implementação.

2 FUNDAMENTAÇÃO TEÓRICA

Este capítulo apresenta a revisão bibliográfica acerca dos temas em que o trabalho está fundamentado. Destacam-se compiladores, a linguagem de consulta SQL e procedimentos armazenados de banco de dados.

2.1 COMPILADORES

Compiladores, de uma forma específica, são tradutores que mapeiam programas escritos em uma linguagem de alto nível em programas equivalentes em linguagem simbólica ou de máquina. No contexto de linguagens de programação, tradutor é um sistema que aceita um programa escrito em uma linguagem de programação, sendo essa a entrada (linguagem fonte) e apresenta como saída um programa equivalente em outra linguagem (linguagem objeto) (PRICE; TOSCANI, 2001, p. 4-5).

Desde o desenvolvimento dos primeiros compiladores no início dos anos 50 até os dias de hoje, o modo de organizar, escrever e utilizar um compilador evoluiu muito. Inicialmente, grande parte dos trabalhos que faziam uso da compilação dedicavam-se a tradução das fórmulas aritméticas para a linguagem de máquina, o que sofreu modificação com o passar dos anos, e transformou-se no uso das linguagens de programação.

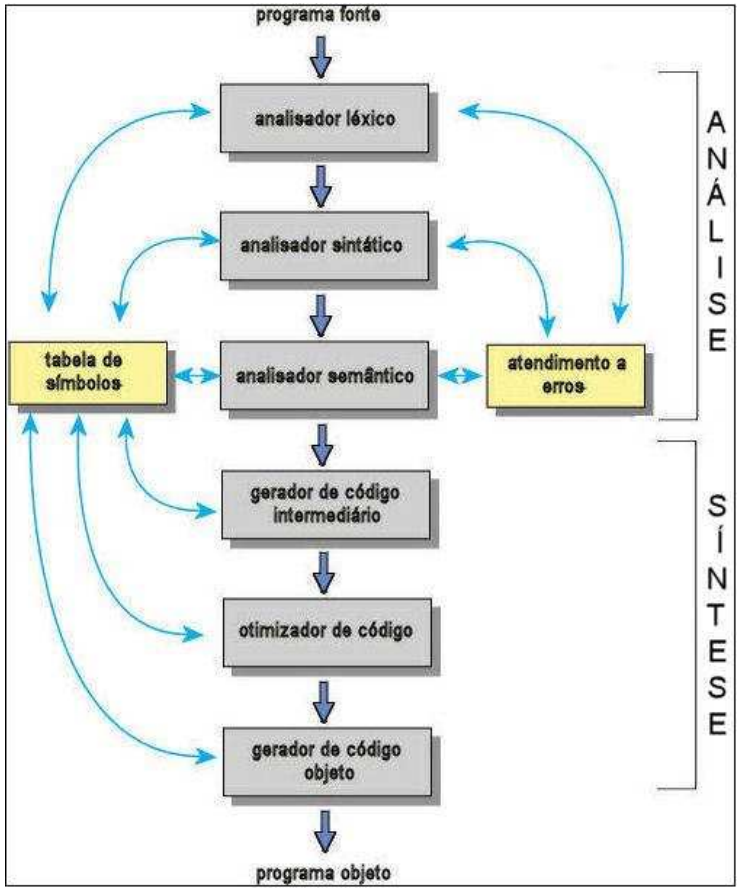
Normalmente, compiladores são programas bastante complexos de se implementar, porém devido ao desenvolvimento de teorias relacionadas às tarefas de análise e síntese de programas, existe uma estrutura básica para o desenvolvimento desse tipo de ferramenta. Constituem-se de funções padronizadas, independentemente da linguagem a ser traduzida ou do resultado a ser gerado, que geralmente compreendem a análise do programa fonte e posterior síntese para produção do código objeto.

2.1.1 As Fases de um Compilador

O processo de compilação de um programa ocorre em fases, sendo que cada uma delas comunica-se com a seguinte por meio de uma linguagem intermediária adequada. Conforme

Aho, Sethi e Ullman (1995, p. 3-5), conceitualmente, cada uma das fases transforma o programa fonte de uma representação para a outra. Na prática, algumas das fases podem ser agrupadas e a representação intermediária entre as mesmas não precisa ser explicitamente construída. Isso ocorre devido ao fato de que as funções básicas para a tradução podem não estar separadas em módulos específicos, podendo apresentar-se em módulos distintos.

Segundo Price e Toscani (2001, p. 6-7), como fases de compilação, pode-se citar: análise léxica, análise sintática, análise semântica, geração de código intermediário, otimização de código, geração de código objeto, gerência de tabela de símbolos e atendimento a erros, como mostra a figura 1.



Fonte: adaptado de Price e Toscani (2001).

Figura 1 – Estrutura de um compilador segundo Price e Toscani

Entretanto vale ressaltar que o processo de tradução pode não envolver todas as fases de compilação. As fases sintáticas otimizador de código e gerador de código objeto, apresentadas na figura 1, podem ser suprimidas em determinadas ocasiões, pois o código intermediário gerado pode ser o código final esperado como saída do processo de tradução.

2.1.2 Análise Léxica

Tem como principal objetivo efetuar a leitura dos caracteres e identificar seqüências logicamente coesivas, definidas como lexemas, que possam constituir uma unidade léxica, ou *token*. À medida que os caracteres são lidos, verifica-se se os mesmos pertencem ao alfabeto da linguagem, desprezando brancos e comentários desnecessários. Os *tokens* constituem classes de símbolos, tais como palavras reservadas, delimitadores, identificadores, que posteriormente serão utilizados pelo *parser* na análise sintática.

Segundo Aho, Sethi e Ullman (1995, p. 39-40), além da leitura e identificação dos caracteres, a análise léxica serve para alimentar a tabela de símbolos, associando o lexema, caso não haja ocorrência do mesmo. O valor léxico associado a um *token* aponta para uma entrada do mesmo na tabela de símbolos, gerando um índice.

Comumente, a implementação do analisador léxico dá-se na forma de sub-rotina ou co-rotina do *parser*, realizando a leitura dos caracteres quando solicitado pelo mesmo, com um comando tipo “obter próximo *token*”.

2.1.3 Autômatos Finitos

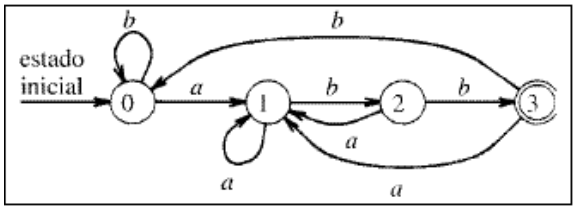
Um autômato finito representa um reconhecedor de linguagens regulares, na forma de um diagrama de transições, utilizado como método para implementação de um analisador léxico, através do reconhecimento de padrões.

Conforme Aho, Sethi e Ullman (1995, p. 51-53), o funcionamento do autômato dá-se através de um modelo matemático para uma máquina de estados finitos. A partir de uma entrada inicial, o autômato realiza a transição entre os estados, na forma de saltos, realizando a análise de cada símbolo, até encontrar o símbolo definido como estado final. Dessa forma, o reconhecimento do *token* dá-se pelo atual estado de transição. Se o estado é final, então o *token* é valido, caso contrário, é inválido. Logo, ao conjunto de todos os *tokens* aceitos pelo autômato chama-se de linguagem aceita pelo autômato.

Os autômatos basicamente dividem-se em dois tipos:

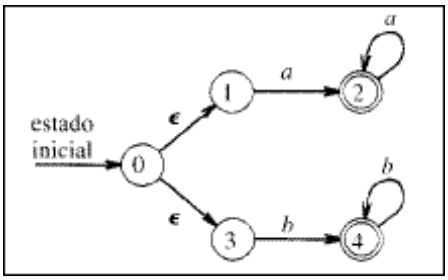
a) Autômato Finito Não-Determinístico (AFN), quando no mínimo um estado possuir mais de uma transição para o mesmo símbolo, conforme a figura 2;
b) Autômato Finito Determinístico (AFD), quando cada estado possuir no máximo

uma transição definida pra cada símbolo, conforme a figura 3.



Fonte: Aho, Sethi e Ullman (1995).

Figura 2 – Um AFN que aceita *(a | b)* abb*



Fonte: Aho, Sethi e Ullman (1995).

Figura 3 – AFD que aceita *aa* | bb**

2.1.4 Análise Sintática

Também definida como análise gramatical, envolve o agrupamento de *tokens* em uma seqüência de símbolos que constituem estruturas sintáticas, como expressões ou comandos, obedecendo às regras gramaticais da linguagem. Como função alternativa a análise sintática, os reconhecedores sintáticos fazem a detecção de erros de sintaxe, identificando de forma clara e objetiva a posição e o tipo do erro ocorrido.

As regras gramaticais que constituem as construções da linguagem podem ser descritas na forma de produções, incluindo entre seus elementos símbolos terminais, que fazem parte do código fonte, e não-terminais, que geram outras regras. Geralmente são apresentadas utilizando a notação *Backus-Naur Form* (BNF).

Através de uma varredura ou *parsing*, o analisador sintático produz uma estrutura em árvore, denominada árvore de derivação. Em Price e Toscani (2001, p. 9-10), é afirmado que em geral, a árvore de derivação não é produzida explicitamente, mas sua construção está implícita nas estruturas recursivas que efetuam a análise sintática.

2.1.5 Gramáticas Livres de Contexto

Uma Gramática Livre de Contexto (GLC) G é uma gramática G = (V, T, P, S), onde V é o conjunto de variáveis, T é o conjunto de terminais, P é o conjunto de produções e S, o símbolo de início, com a restrição de que qualquer regra de produção de P é da forma $A \rightarrow \alpha$, onde A é uma variável de V e α uma palavra de $(V \cup T)^*$. Logo, uma GLC é uma gramática onde o lado esquerdo das produções contém exatamente uma variável (HOPCROFT; ULLMAN; MOTWANI, 2002, p. 180-184).

O que caracteriza a GLC é a propriedade de que o símbolo não-terminal A pode ser substituído pela cadeia α do lado direito da regra, onde quer que A ocorra, independentemente do contexto. Em outras palavras, um símbolo não terminal pode ser substituído por sua produção gramatical, sem fazer qualquer análise dos símbolos sucessores ou antecessores ao não-terminal. No quadro 1 observa-se um exemplo de uma GLC e uma expressão regular definida pela mesma.

```
S -> 0B | 1C
B -> 1S
C -> 0C | 0D | 1
D -> 0

V = {S, B, C, D}

T = {0, 1}

S

ER: (01)*10*(00|1)
```

Quadro 1 – Exemplo de uma GLC G = (V, T, P, S)

Conforme Price e Toscani (2001, p. 30), as GLC formam a base para a análise sintática das linguagens de programação, pois permitem descrever a maioria das linguagens utilizadas atualmente.

Hopcroft, Ullman e Motwani (2002, p. 182-183) definem quatro componentes importantes em uma descrição gramatical de uma linguagem:

a) existe um conjunto finito de símbolos que formam os *tokens* da linguagem, definidos como símbolos terminais;
b) existe um conjunto finito de variáveis, chamadas de não-terminais ou categorias sintáticas, que são os símbolos utilizados na descrição da linguagem, onde cada variável representa um conjunto de *tokens*;

c) uma das variáveis representa a linguagem que está sendo definida, denominada símbolo de início;
d) existe um conjunto finito de produções ou regras, que representam a definição recursiva da linguagem, especificando a forma pela qual os terminais e não-terminais podem ser combinados, sendo que cada produção contém exatamente um símbolo não terminal à esquerda.

Em José Neto (1987, p. 48-49), é afirmado que uma metalinguagem comumente utilizada para expressão da sintaxe de uma linguagem de programação é a BNF. Trata-se de uma notação recursiva de formalização da sintaxe de linguagens através de produções gramaticais, permitindo assim a criação de dispositivos de geração de sentença. Através do uso da BNF, e restringindo-se ao lado esquerdo das produções à forma de um não-terminal isolado, é possível representar, de modo recursivo, qualquer linguagem livre de contexto.

A simbologia adotada na notação BNF é a seguinte:

a) <x> representa um símbolo não-terminal, onde x é o nome do não-terminal;
b) a seqüência <x> ::= β representa uma regra de produção, associando o não-terminal <x> à sentença β, sendo que <x>::= β significa "<x> é definido por β";
c) o caractere | separa as diversas regras de produção que estão à direita do símbolo ::=, desde que o símbolo não-terminal à esquerda seja o mesmo;
d) o significado de <x> ::= β1 | β2 | ... | βn é <x> é definido por β1 OU <x> é definido por β2 OU ... <x> é definido por βn;
e) o símbolo x ou X representa um símbolo terminal, dado pela cadeia x ou X de caracteres quaisquer e deve ser escrito tal como aparece nas sentenças da linguagem;
f) o símbolo ε significa vazio.

No quadro 2 observa-se a utilização da BNF na notação da expressão IF-THEN-ELSE da GLC especificada no Apêndice A. A estrutura *if*, definida a partir do não-terminal <ifstmt>, deve possuir no mínimo um identificador (IF) , seguido de uma condição para execução do bloco (<search-condition>), sucedida por outro identificador (THEN). Em seguida, os não-terminais <new-stmt> e <else> representam a estrutura do bloco de comandos, onde <stmt> define uma única instrução e <stmt-block> define um bloco de instruções. O não-terminal <else> é definido com um identificador (ELSE) seguido do não terminal <new-stmt>, que definirá um bloco de comandos, ou poderá ser não existir, definido pelo vazio (ε).

```
<ifstmt> ::= IF <search-condition> THEN <new-stmt> <else>
<new-stmt> ::= <stmt> | <stmt-block>
<else> ::= ELSE <new-stmt> | ε
```

Quadro 2 – Exemplo de uma GLC utilizando a notação BNF

A expressão de uma GLC na forma gráfica dá-se através de uma árvore de derivação. Segundo Price e Toscani (2001, p. 31-32), essa representação apresenta, de forma explícita, a estrutura hierárquica que originou a sentença. A raiz dessa árvore é o símbolo inicial da gramática. Cada vértice interior da árvore é rotulado por um não-terminal. Símbolos terminais e a palavra vazia são vértices folha.

Derivando-se sempre o símbolo não-terminal mais à esquerda, tem-se uma derivação mais à esquerda. Já uma derivação mais à direita aplica sempre as produções ao não-terminal mais à direita.

2.1.6 Análise Descendente (*Top-down*)

Conforme José Neto (1987, p. 57), na análise descendente seleciona-se para substituição o não-terminal mais à esquerda que figura na forma sentencial, com leitura da cadeia de entrada da esquerda para a direita, sendo essa a pré-ordem da árvore de derivação. A árvore então cria a raiz e, a seguir, cria subárvores filhas, da esquerda para a direita. Assim é produzida a derivação mais a esquerda da sentença em análise. Destacam-se três formas para implementação de um *parser* top-down: recursivo com retrocesso (*backtracking*), recursivo preditivo e tabular preditivo.

Na técnica de análise recursiva com retrocesso, cada símbolo não-terminal é implementado por um procedimento que efetua o reconhecimento do lado direito das produções que definem o símbolo, normalmente utilizando recursividade, na forma de uma pilha ímplicita. A expansão da árvore de derivação ocorre a partir da raiz, expandindo sempre o não-terminal mais a esquerda. Caso exista mais de uma regra de produção para o não-terminal a ser expandido, o *token* sob o cabeçote de leitura será utilizado. Se o *token* de entrada não define a produção, todas as alternativas serão utilizadas até que se obtenha êxito, ou que a análise falhe complemente (PRICE;TOSCANI, 2001, p. 38).

Há uma técnica de análise recursiva implementada sem a utilização de retrocesso. Assim como na análise recursiva com retrocesso, na análise recursiva preditiva procedimentos

são implementados para cada símbolo não-terminal. Nesse tipo de analisador, o símbolo sob o cabeçote determina exatamente qual produção deve ser aplicada na expansão para cada não-terminal. São comumente denominados analisadores descendentes do tipo *Left to Right with Leftmost Derivation* (LL(1)).

Entretanto a gramática precisa suprir algumas exigências feitas pelo analisador recursivo preditivo: que a gramática não tenha recursividade à esquerda, que a gramática esteja fatorada à esquerda e que, para os não-terminais com mais de uma regra de produção, os primeiro terminais deriváveis sejam capazes de identificar, univocamente, a produção que deve ser aplicada a cada instante da análise.

A análise preditiva tabular consiste em uma técnica de análise não recursiva, fazendo uso de uma pilha explícita ao invés de chamadas recursivas. De acordo com Price e Toscani (2001, p. 45), esse analisador implementa um autômato de pilha controlado por uma tabela de análise. O princípio do reconhecimento preditivo é a determinação da produção a ser aplicada, cujo lado direito irá substituir o símbolo não-terminal que consta naquele momento no topo da pilha. A produção a ser aplicada é buscada levando em conta o não-terminal no topo da pilha e o *token* sob o cabeçote de leitura.

2.1.7 Tradução Dirigida por Sintaxe

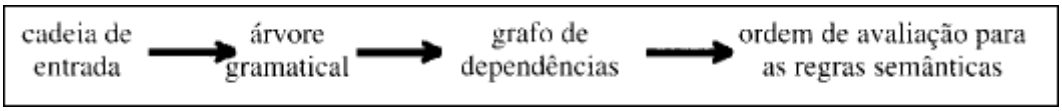
A tradução dirigida por sintaxe é uma técnica que une o processo de análise sintática com a fase de geração de código, fazendo uso da associação de ações semânticas às regras de produção da gramática, atrelando variáveis a símbolos terminais e não-terminais da gramática. Dessa forma, os símbolos passam a ter atributos ou parâmetros que podem realizar o armazenamento de valores na execução do processo de reconhecimento.

Em Aho, Sethi e Ullman (1995, p. 120), é afirmado que a execução de uma regra semântica associada pode resultar em ações como geração de código, armazenamento de informações na tabela de símbolos, emitir mensagens de erro ou realizar quaisquer outra atividade.

Conceitualmente existem duas notações para associar uma ação semântica a uma regra de produção: definições dirigidas pela sintaxe e esquemas de tradução. Independentemente da notação utilizada, o processo como um todo compreende a análise do fluxo de *tokens* de entrada, construção da árvore gramatical e o percorrimento da árvore, avaliando as ações semânticas de cada nó.

Um esquema de tradução é a extensão de uma GLC, feita através da associação de atributos aos símbolos gramaticais e de ações semânticas às regras de produção, indicando a ordem das operações a serem realizadas pelo compilador no processo de análise sintática.

Em uma definição dirigida por sintaxe, cada símbolo gramatical tem associado a si um conjunto de atributos, separados em dois subconjuntos, definidos como herdados ou sintetizados. Segundo Price e Toscani (2001, p. 86-87), sendo S um símbolo, o valor de um atributo de S é denominado sintetizado se ele é computado a partir, exclusivamente, dos valores dos atributos dos filhos de S, na árvore de derivação. Já o valor de um atributo S é dito herdado se ele é computado a partir dos valores dos atributos dos irmãos ou do pai de S. Na figura 4, pode-se observar a visão conceitual da tradução dirigida por sintaxe.



Fonte: Aho, Sethi e Ullman (1995).

Figura 4 – Visão conceitual da tradução dirigida por sintaxe

Devido ao fato de que regras semânticas computam valores de atributos a partir de outros atributos, há uma dependência estabelecida entre as mesmas, considerando que em algumas ocasiões um atributo necessita que outros estejam calculados previamente, antes de ser calculado. Logo, faz-se necessária uma ordenação correta dos cálculos.

De acordo com Aho, Sethi e Ullman (1995, p. 122-123), as interdependências entre os atributos herdados e sintetizados nos nós da árvore gramatical podem ser delineadas através de um grafo chamado grafo de dependências.

Entretanto, antes de construir um grafo de dependências para uma árvore de derivação, cada regra semântica deve estar sob a forma $b := f(c_1, c_2, ..., c_k)$, onde b é um atributo sintetizado fictício. Para cada atributo o grafo deve conter um nó, e terá um arco dirigido do atributo c para o atributo b, se b depender de c. A construção dá-se utilizando o algoritmo do quadro 3.

```
para cada nó N da árvore de derivação faça:
   para cada atributo a de N faça:
      construa um nó para representar o atributo a.
para cada nó N na árvore de derivação faça:
   para cada regra semântica b := f(c1, c2,...,ck) associada ao nó N faça:
      para i := 1 até k faça:
         construa um arco do nó c para o nó b.
```

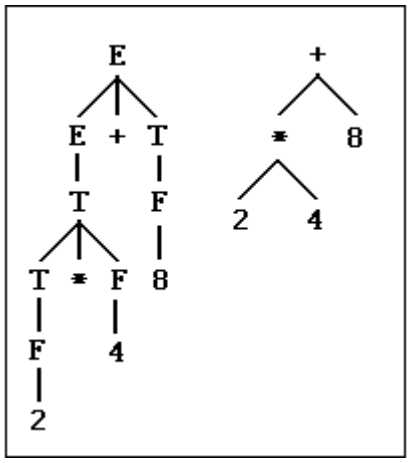
Fonte: adaptado de Price e Toscani (2001).

Quadro 3 – Algoritmo para definição de atributos sintetizados e herdados num grafo de dependência

2.1.8 Árvores de Sintaxe

A árvore de sintaxe é uma forma condensada dá árvore de derivação, na qual somente os operandos da linguagem aparecem como folhas; os operadores passam a ser nós interiores da árvore. Outra simplificação encontrada nas árvores sintáticas é que cadeias de produções singelas, como A → B ou B → C, podem ser eliminadas. O uso de árvores de sintaxe como representação intermediaria da sentença de entrada permite que a tradução para código objeto seja desligada da análise.

Conforme Price e Toscani (2001, p. 96-98), para construção de uma árvore de sintaxe um nó operador deve possuir três campos: um para identificar o operador, e outros dois para indicar os nós de operandos. Podem ainda existir campos adicionais para outros atributos associados aos nós, se for o caso. Na figura 5 representa uma árvore de derivação para uma sentença dada.



Fonte: adaptado de Price e Toscani (2001).

Figura 5 – Árvore de derivação e árvore de sintaxe para a sentença 4*2+8

2.1.9 Geração de Código

De acordo com Price e Toscani (2001, p. 96-98), o código intermediário é a transformação da árvore de derivação em um segmento de código, que pode ser, eventualmente, o código objeto final. Essa forma destaca-se como uma boa alternativa, visto que a manipulação das árvores é relativamente simples, permitindo aos algoritmos de otimização efetuarem simplificações substanciais no programa através da eliminação de redundâncias e da reorganização das operações indicadas.

Conforme Aho, Sethi e Ullman (1995, p. 222-223), existem várias exigências impostas a um gerador de código. O código de saída precisa ser correto e de alta qualidade, significando que o mesmo deve fazer uso adequado dos recursos da máquina-alvo. A familiaridade com a máquina-alvo e seu conjunto de instruções também contribuem para o desenvolvimento de um bom gerador de código.

2.2 LINGUAGENS DE BANCO DE DADOS

Segundo Ramakrishnan e Gehrke (1999, p. 51-52), um modelo de banco de dados relacional é baseado em entidades e relacionamentos, conforme definido por E. F. Codd em 1969, utilizando a percepção do mundo real. Uma entidade é qualquer pessoa distinguível, lugar, ou coisa que é de interesse por alguma razão, podendo ser algo concreto ou abstrato. Um relacionamento é uma associação entre entidades. Um modo interessante de representar um modelo relacional é fazendo o uso do conceito de matrizes. As linhas da matriz seriam os registros de uma tabela e as colunas os campos.

Para manipular os dados e estruturas de controle das tabelas de um banco de dados, fez-se necessário o desenvolvimento de uma linguagem que oferecesse suporte a consulta de dados, promovendo a utilização dos relacionamentos entre as tabelas. Através dos relacionamentos, pode-se selecionar somente dados que atendam as necessidades específicas de uma determinada consulta.

Diante da necessidade, a linguagem de banco de dados SQL foi definida e introduzida nos bancos de dados relacionais, sendo que a mesma é apresentada na primeira seção deste capítulo. Em seguida, na segunda seção, faz-se uma abordagem sobre as estruturas internas de

um banco de dados, chamados procedimentos armazenados, assim como da extensão do SQL para desenvolvimento dessas estruturas.

2.2.1 A Linguagem SQL

Em 1974 surgiu a linguagem *Structured English Query Language* (SEQUEL) nos laboratórios da *International Business Machines Corporation* (IBM), e no ano seguinte foi implementado um protótipo da mesma. Entre os anos de 1976 e 1977 foi definida uma versão revisada do SEQUEL, chamada SEQUEL/2, que logo após passou a chamar-se SQL. No final dos anos 70 e início dos anos 80 a linguagem SQL foi implementada em alguns produtos da própria IBM e de outros fabricantes que passaram a suportar essa linguagem (SQL, 2006). No quadro 4, verifica-se a evolução da padronização da linguagem SQL.

Ano	**Nome**	**Apelido**	**Comentários**
1986	SQL-86	SQL-87	Primeiramente publicada pelo *American National* Standards *Institute* (ANSI). Retificado pelo *International Standards Organization* (ISO) em 1987.
1989	SQL-89		Revisão menor.
1992	SQL-92	SQL2	Revisão maior.
1999	SQL-1999	SQL3	Adicionado expressões regulares de emparelhamento, consultas recursivas, gatilhos (*triggers*), tipos não escalados e características da orientação a objeto.
2003	SQL-2003		Introduz características relacionadas a XML, seqüências padronizadas e colunas com valores de auto-generalização.

Fonte: adaptado de SQL (2006).

Quadro 4 – Evolução da padronização da linguagem SQL

De acordo com Groff e Weinberg (1999, p. 9-10), a linguagem SQL é uma ferramenta para organizar, gerenciar e extrair dados armazenados em um banco de dados, sendo ao mesmo tempo extremamente poderosa e fácil de aprender. Consiste em uma linguagem de consulta declarativa, baseada em álgebra relacional. Assim, é utilizada em SGBDs que utilizam o modelo relacional.

2.2.2 Sub-linguagens

Ramakrishnan e Gehrke (1999, p. 119) definem sub-linguagens internas à linguagem SQL, utilizadas para manipulação de dados e criação de consultas, sendo elas:

a) *Data-Definition Language* (DDL): usada para criação, com o comando *create*, remoção, através do comando *drop* e alteração de objetos internos do SGBD, utilizando o comando *alter*. Grande parte dos SGDBs comerciais tem extensões proprietárias do DDL;
b) *Data-Manipulation Language* (DML): possibilita ao usuário a construção de consultas utilizando o comando *select*, inserção de dados através do comando *insert*, modificação de dados existentes utilizando o comando *update* e remoção de dados, através do comando *delete*. Destacam-se ainda os comandos *begin transaction*, que inicializa uma transação, *commit*, que encerra a transação, gravando permanentemente os dados no SGBD e *rollback*, que encerra a transação, descartando, porém, as alterações feitas pelo usuário;
c) *Data-Query Language* (DQL): usada para criação de consultas, agregando apenas o comando *select*.

2.2.3 Tipos de Dados

Conforme Groff e Weinberg (1999, p. 58-60), os padrões ANSI e ISO especificam os vários tipos de dados que podem ser armazenados em um SGBD baseado e manipulado pela linguagem SQL. Entretanto, SGBDs comerciais geralmente possuem diferenças entre os tipos de dados, sendo uma das maiores barreiras para a portabilidade. Essas diferenças têm origem na inovação tecnológica dos bancos de dados relacionais. Após um determinado tempo, o novo tipo de dado é padronizado, e lentamente começa a tornar-se padrão em outros SGBDs.

Um tipo de dado define-se como um conjunto de valores representáveis, sendo que um tipo é dito primitivo quando não há subdivisão lógica. O quadro 5 mostra os tipos de dados SQL e que tipos lógicos são representados.

Tipos SQL	Descrição
Character	Cadeias de caracteres com tamanho fixo
Character varying	Cadeias de caracteres com tamanho variável, podendo informar o tamanho máximo
Bit	Cadeias de bits com tamanho fixo
Bit varying	Cadeias de bits com tamanho variável, podendo informar o tamanho máximo
Numeric	Numéricos, inteiros ou decimais, exatos
Decimal	Numéricos decimais exatos
Integer	Numéricos inteiros exatos
Smallint	Numéricos inteiros exatos com tamanho pequeno
Float	Numéricos aproximados
Real	Numéricos aproximados
Double precision	Numéricos aproximados com tamanho ou precisão grande
Date	Data
Time	Hora
Timestamp	Data/Hora
Interval	Intervalo

Fonte: Hiebert (2003).

Quadro 5 – Tipos de dados SQL e representação lógica

2.3 PROCEDIMENTOS ARMAZENADOS

São procedimentos fisicamente armazenados dentro de um SGBD, sendo um recurso disponível nos principais bancos de dados relacionais. Comumente apresentam diferenças em sua implementação, sofrendo variação de um SGBD para outro. Normalmente, essa diferença tem origem na linguagem estendida e embutida que o SGBD possui em sua implementação para dar suporte ao desenvolvimento dos procedimentos.

O funcionamento de um procedimento armazenado dá-se como uma função qualquer em uma linguagem de programação. Segundo Sunderic e Woodhead (2000, p. 32), em um procedimento armazenado pode-se fazer uso de variáveis, constantes, tipos de dados, parâmetros de entrada e saída, valores de retorno, execuções condicionais, estruturas de repetição, comentários e comandos de comparação.

Um procedimento armazenado é classificado como um *Shared Database Object* (SDO), ou seja, um objeto compartilhável de banco de dados. O objetivo de um SDO é fornecer um único procedimento que pode ser acessado constantemente por qualquer usuário do banco. Dessa forma, não há a necessidade de procedimentos que possuam a mesma

funcionalidade, evitando assim a propagação de objetos idênticos, o que acarretaria em uma redundância.

No quadro 6, observa-se um procedimento armazenado desenvolvido seguindo as regras da linguagem definida no Apêndice A.

```
PROCEDURE XPTO (aIndex IN Integer, aData IN Timestamp, aRet OUT Char(2)) AS
  Declare xExiste Timestamp
BEGIN
  --verifica se existe registro no indice, na tabela Tab1
  select datins into xExiste from Tab1 where position = aIndex
  --seleciona de Tab2 a quantidade limite de registros em Tab1
  --e o proximo valor a ser inserido em Tab1
  if xExiste <> aData then
    --se a data for diferente, atualiza
    update Tab1 set position = xCurPos, datins = aData  where position = aIndex
  else
    --se nao existir, insere
    insert into Tab1 (position, datins) values (xCurPos, aData)
  set aRet = 'OK'
END
```

Quadro 6 – Exemplo de procedimento armazenado utilizando a linguagem definida pela GLC do Apêndice A

Quando se faz a chamada a um procedimento, deve-se verificar se a passagem de parâmetros está sendo feita da forma adequada, caso sejam necessários. Se um parâmetro for de entrada (*IN*), seu valor apenas poderá ser usado pelo procedimento. Se for de saída (*OUT*), não deve ser passado nenhum valor, porém o procedimento deverá retornar um valor nesse parâmetro. Um parâmetro pode ainda ser de entrada e saída (*INOUT*), sendo que deverá ser passado um valor para esse parâmetro e o mesmo ainda terá um valor retornado pelo procedimento.

Embora haja divergência entre programadores quanto ao uso de procedimentos armazenados, é importante ressaltar que existem algumas vantagens na utilização desse recurso, entre as quais destacam-se:

a) pré-compilação de consultas: as consultas contidas em um procedimento armazenado já foram compiladas pelo SGBD, portanto possuem um plano específico definido antes de sua execução, reduzindo o *overhead* que ocorre em uma consulta normal;
b) execução no servidor: um procedimento armazenado executa diretamente no SGBD, tendo acesso direto aos dados necessários na execução das tarefas, o que é mais rápido do que realizar consultas sucessivas de fora do servidor, evitando tráfego de rede no retorno das consultas;
c) outros usos: pode ser utilizado para fazer o controle de transação, ou ainda,

executar em um horário previamente definido pelo administrador do SGBD, evitando assim, execução em horários de pico.

Os procedimentos armazenados no SGBD Oracle, descritos em Lakshman (2003), são implementados utilizando a linguagem PL/SQL, que é uma extensão procedural da linguagem SQL, definida como linguagem de quarta geração, incorporando recursos estruturais de uma linguagem de terceira geração.

A linguagem PL/SQL é uma linguagem orientada a blocos, que por sua vez são divididos nas seções de declaração, execução e tratamento de erros. A primeira seção é opcional, pois podem haver procedimentos que não necessitem declaração de variáveis. A terceira seção também é considerada opcional, levando em conta que o tratamento de exceções, apesar de ser uma boa prática de programação, é dispensável. Já a segunda seção, localizada entre as palavras reservadas *begin* e *end*, é o corpo do procedimento.

Na obra de Petkovic (2001, p. 155-160), é afirmado que a linguagem de definição de dados no SGBD MS SQL Server é a Transact-SQL. Tal como em outras linguagens embarcadas, essa linguagem estende o SQL padrão, proporcionando recursos específicos para utilização no SGBD hospedeiro.

Na Transact-SQL, todas as variáveis são identificadas pelo "@" inicial. Os comandos devem sempre iniciar com uma palavra reservada, sendo que a linguagem não apresenta um caractere finalizador de comandos, diferentemente da PL/SQL, onde os comandos são finalizados por ";". Outra diferença presente na Transact-SQL é a ausência da seção de declaração de variáveis. As mesmas são declaradas no corpo do procedimento. Há ainda a possibilidade de se criar um procedimento sem que os objetos que o mesmo referencia (tabelas, por exemplo) existam.

Em Geschwinde e Schönig (2002, p. 131-132) é afirmado que o SGBD PostgreSQL apresenta uma linguagem própria para desenvolvimento de procedimentos denominada *Procedural Language/Postgre Structured Query Language* (PL/pgSQL), bastante similar a PL/SQL utilizada pelo SGBD Oracle, porém com recursos específicos para o PostgreSQL.

Uma diferença extremamente significativa para a PL/SQL é o fato de na PL/pgSQL não existirem *procedures*, formalmente falando, sendo que todo procedimento armazenado será definido como *function*, independentemente do retorno e tipos de parâmetro. Porém, internamente o banco classificará a "função" criada como sendo uma *procedure*. Para procedimentos que não possuam parâmetro de retorno (*INOUT* ou *OUT*), deve-se, após a declaração dos parâmetros fazer uso da definição "*RETURNS void*", indicando ao PostgreSQL que esse procedimento não terá retorno.

2.4 TRABALHOS CORRELATOS

Hiebert (2003) desenvolveu um protótipo de compilador para a linguagem PL/SQL, possibilitando a geração de código fonte para diferentes SGBDs, sendo esses o Oracle e o MS SQL Server. O desenvolvimento do protótipo deu-se utilizando o ambiente de programação Delphi 5, definindo estruturas de dados conforme a definição da linguagem na notação BNF.

Em seu trabalho, Hiebert (2003) define extensões e complementos para a ferramenta, possibilitando o incremento da geração de código para outros SGBDs e extensão da gramática da linguagem padrão, promovendo o tratamento de funções de banco de dados não implementadas em sua gramática original.

3 DESENVOLVIMENTO DO TRABALHO

Neste capítulo estão apresentadas informações detalhadas inerentes à especificação e implementação do software. São descritos os requisitos do sistema, os diagramas criados para representá-lo, como foi realizada a implementação do software e quais as principais conclusões e limitações.

3.1 REQUISITOS PRINCIPAIS DO PROBLEMA A SER TRABALHADO

A ferramenta desenvolvida nesse trabalho atendeu aos seguintes quesitos:

a) a aplicação deverá ter um espaço definido para escrita do código fonte (Requisito Funcional - RF);
b) a aplicação deverá executar as análises léxica e sintática, a fim de identificar possíveis erros ou inconsistências, apontando os mesmos de forma concisa (RF);
c) a aplicação deverá permitir a geração do código objeto, oferecendo a possibilidade de geração para cada um dos SGBDs (RF);
d) a aplicação deverá permitir ao usuário salvar o código fonte em um arquivo no formato texto, com extensão .lp (RF);
e) a aplicação deverá possibilitar a abertura de um arquivo já salvo na extensão .lp, para edição ou geração de código (RF);
f) a aplicação deverá permitir a conexão com cada um dos SGBDs, possibilitando a criação do procedimento armazenado (RF);
g) a linguagem procedural proposta por Hiebert (2003) deve ser estendida, implementando as funções coalesce, to_char, to_date, to_number, cast, lower, upper, subst, length, pos; cláusulas order by e group by; estrutura de cursores e looping (RF);
h) a aplicação deverá utilizar como linguagem para o código fonte, a extensão da linguagem procedural proposta por Hiebert (2003, p. 49-52) (Requisito Não-Funcional – RNF);
i) a aplicação deve ser desenvolvida utilizando a ferramenta Delphi 7 (RNF).

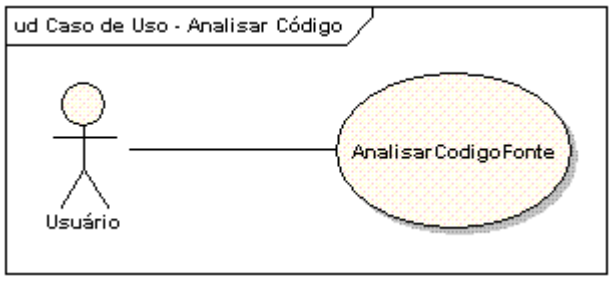
3.2 ESPECIFICAÇÃO

Para realizar a especificação do aplicativo foram utilizados os conceitos da linguagem *Unified Modeling Language* (UML) que segundo Larman (2001, p. 10) “é uma linguagem para especificar, visualizar, construir e documentar os artefatos de um sistema de software, assim como para modelos de negócios e para sistemas não-software”. Para a modelagem foi utilizada a ferramenta Enterprise Architect (SPARX SYSTEMS, 2006) e utilizados quatro diagramas da UML: diagrama de caso de uso, diagrama de classes, diagrama de atividades e diagrama de seqüências.

3.2.1 Diagramas de caso de uso

Os diagramas de caso de uso, segundo Larman (2001, p. 48), servem para definir requisitos funcionais, que devem indicar o que o sistema irá fazer ou o modo como ele irá se comportar diante de uma determinada ação. A UML define os diagramas de caso de uso para ilustrar o nome dos casos, atores e o relacionamento entre ambos.

No quadro 7 pode-se verificar o diagrama de casos de uso para a ação AnalisarCodigoFonte, ou seja, realizar a compilação do código fonte para verificação de erros. Esse caso apresenta três restrições e três cenários, detalhados nos quadros 8 e 9, respectivamente.



Quadro 7 – Caso de uso analisar código

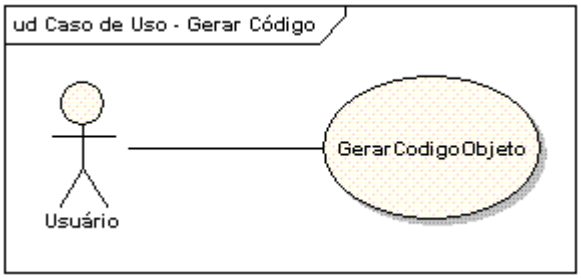
Restrição	Tipo	Status	Detalhes
Fonte	Pré-Condição	Aprovada	Recebe o programa fonte
Análise	Processamento	Aprovada	Realiza as análises léxica e sintática.
Saída	Pós-Condição	Aprovada	Se o programa estiver correto, ou seja, se não contiver erros então retorna a árvore de sintaxe, senão retorna a lista de erros.

Quadro 8 – Restrições do caso de uso analisar código

Cenário	Tipo	Detalhes
Analisar Arquivos	Principal	1.1) O sistema está em sua tela inicial, possibilitando ao usuário abrir um arquivo salvo previamente, através do botão "Abrir" ou digitar o código fonte. 1.2) Ao clicar no botão "Compilar", a ferramenta deverá fazer a análise léxica e sintática do código fonte.
Verificar Resultado	Alternativo	2.1) Se no item 1.1 o usuário escolher a opção "Compilar", será exibida a mensagem de sucesso na operação.
Erro de Análise	Pós-Condição	3.1) Se no item 1.1 o usuário clicar em "Compilar", e o código fonte apresentar erros, os mesmos serão informados ao usuário.

Quadro 9 – Cenários do caso de uso analisar código

No quadro 10 observa-se o diagrama de casos de uso para a ação GerarCodigoObjeto, que realiza explicitamente a compilação do código fonte, apontando erros de forma coerente, se encontrados. Esse caso apresenta três restrições e três cenários, detalhados nos quadros 11 e 12, respectivamente.



Quadro 10 – Caso de uso gerar código

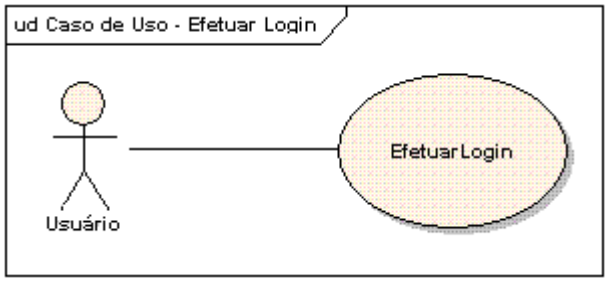
Restrição	Tipo	Status	Detalhes
Entrada	Pré-Condição	Aprovada	Recebe a árvore sintática.
Geração	Processamento	Aprovada	Efetua a geração de código para o SGBD Oracle, para o SGBD MS SQL Server e para o SGBD PostgreSQL, a partir da árvore sintática.
Saída	Pós-Condição	Aprovada	Se a árvore sintática estiver correta, ou seja, se não contiver erros então retorna o código objeto, senão retorna a lista de erros.

Quadro 11 – Restrições do caso de uso gerar código

Cenário	Tipo	Detalhes
Analisar Arquivos	Principal	1.1) O sistema está em sua tela inicial, possibilitando ao usuário abrir um arquivo salvo previamente, através do botão "Abrir" ou digitar o código fonte. 1.2) Ao clicar no botão "Gerar Código", a ferramenta deverá fazer a análise léxica e sintática do código fonte e gerar o código objeto.
Verificar Resultado	Alternativo	2.1) Se no item 1.1 o usuário escolher a opção "Gerar Código", o código é gerado pela ferramenta será exibida a mensagem de sucesso na operação.
Erro de Análise	Pós-Condição	3.1) Se no item 1.1 o usuário clicar em "Gerar Código", e o código fonte apresentar erros, os mesmos serão informados ao usuário.

Quadro 12 – Cenários do caso de uso gerar código

No quadro 13 é exibido o diagrama de casos de uso para a ação EfetuarLogin, que realiza a conexão com o SGBD desejado, mediante informação dos dados referentes a usuário, senha e servidor. Esse caso apresenta três restrições e três cenários, detalhados nos quadros 14 e 15, respectivamente.



Quadro 13 – Caso de uso efetuar login

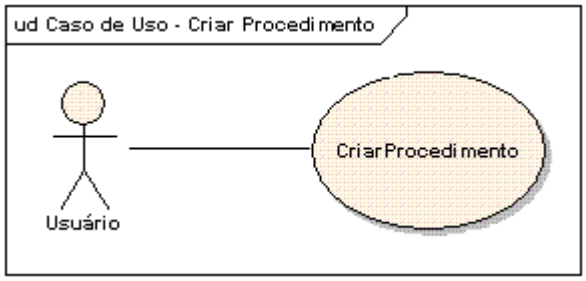
Restrição	Tipo	Status	Detalhes
SGDB Ativo	Pré-Condição	Aprovada	O banco de dados deve estar ativo previamente.
Login	Processamento	Aprovada	O sistema deverá realizar a conexão com o SGDB.
Saída	Pós-Condição	Aprovada	Se os dados informados estiverem corretos e o SGDB ativo, a conexão deverá ser efetuada, caso contrário deverá ser ser exibida a mensagem de erro.

Quadro 14 – Restrições do caso de uso efetuar login

Cenário	Tipo	Detalhes
Realizar Conexão	Alternativo	1.1) O sistema está na tela que possui o código gerado para determinado SGDB. 1.2) Se no item 1.1 o usuário clicar em "Conectar", a tela de login deverá ser exibida ao usuário 1.3) Se no item 1.1 o usuário clicar em "Conectar" e já houver conexão com o SGDB, a mesma será encerrada.
Efetuar Login	Alternativo	2.1) Se no item 1.1 o usuário escolheu a opção "Conectar", a tela de login será exibida, exigindo Servidor, Login e Senha. 2.2) Se no item 2.1 o usuário preecher todos os campos e clicar em "OK", a conexão com o banco de dados deverá ser realizada.
Erro no Login	Exceção	3.1) Se no item 2.1 o usuário informar um login inválido, a aplicação deverá exibir a mensagem de erro na autenticação. 3.2) Se no item 2.1 o usuário informar um login válido e ocorrer um erro qualquer de conexão, o mesmo deverá ser exibido ao usuário.

Quadro 15 – Cenários do caso de uso efetuar login

No quadro 16 observa-se o diagrama de casos de uso para a ação CriarProcedimento, que cria o procedimento armazenado no SGBD. Esse caso apresenta quatro restrições e três cenários, detalhados nos quadros 17 e 18, respectivamente.



Quadro 16 – Caso de uso criar procedimento

Restrição	Tipo	Status	Detalhes
Conexão	Pré-Condição	Aprovada	A ferramenta deve ter conexão estabelecida com o SGBD.
Código	Pré-Condição	Aprovada	O código objeto deve estar gerado.
Criação	Processamento	Aprovada	Realiza a criação do procedimento armazenado.
Saída	Pós-Condição	Aprovada	Se o código gerado estiver adequado a semântica do banco, o mesmo deve ser criado com sucesso. Caso contrário, o erro deverá ser exibido.

Quadro 17 – Restrições do caso de uso criar procedimento

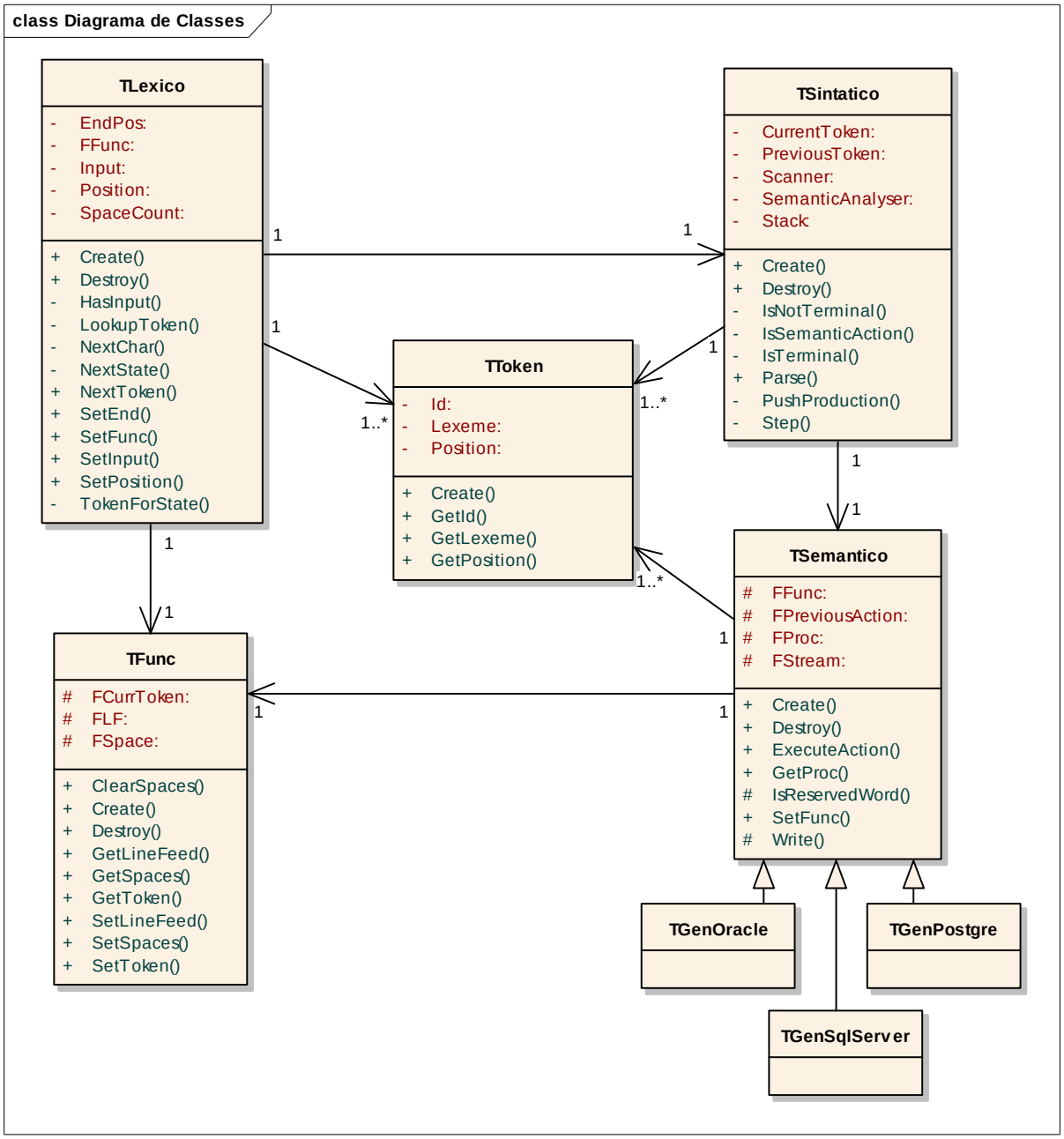
Cenário	Tipo	Detalhes
Criar Procedimento	Alternativo	1.1) O sistema está com o código objeto já gerado e conectado ao SGDB. 1.2) Ao clicar no botão "Criar", a criação do procedimento armazenado deve ser realizada. 1.3) Se no item 1.1 o usuário clicar em "Salvar", o código gerado deve ser salvo.
Verificar Resultado	Alternativo	2.1) Se no item 1.1 o usuário escolher a opção "Criar", a mensagem de sucesso deve ser exibida ao usuário. 2.2) Se no item 1.1 o usuário clicar em "Salvar", o código gerado deve ser salvo.
Erro na Criação	Exceção	3.1) Se no item 1.1 o usuário escolher a opção "Criar", e o SGBD retornar erro na criação, uma mensagem contendo o erro deve ser exibida ao usuário.

Quadro 18 – Cenários do caso de uso criar procedimento

3.2.2 Diagrama de classes

Conforme Larman (2001, p. 286), um diagrama de classes na UML é utilizado para

ilustrar as especificações das classes da aplicação. Tipicamente apresenta informações sobre classes, com suas associações e atributos, interfaces, contendo operações e constantes, métodos, tipos de atributo, navegabilidade e dependências. Pode-se observar no quadro 19 o diagrama de classes da aplicação desenvolvida.



Quadro 19 – Diagrama de classes

O analisador léxico é representado pela classe TLexico, sendo que a mesma tem como objetivo principal retornar *tokens*, ou objetos do tipo TToken, para o analisador sintático através do método NextToken. Ao encontrar uma palavra, o analisador verifica se o *token* pertence à linguagem, através de uma tabela de *tokens*, usando o método TokenForState. Se o

resultado for diferente de 0, então o *token* é válido. No atributo Input fica armazenado o código fonte. Já no atributo Position é guardada a posição atual do analisador. O atributo EndPos é utilizado no reconhecimento dos *tokens*, indicando a posição final do mesmo. O atributo SpaceCount armazena a quantidade de espaços entre as palavras, atualizando a instância de TFunc representada pelo atributo FFunc.

O analisador sintático é exibido pela classe TSintático. Os atributos CurrentToken e PreviousToken armazenam, respectivamente, o *token* atual e o anterior. O atributo Scanner é uma instância do tipo TLexico, ou seja, o analisador léxico. O atributo SemanticAnalyzer é uma instância do analisador semântico, utilizada para processar as ações semânticas durante a execução da análise sintática. A lista de produções é representada pelo atributo Stack, que é utilizado pelo método PushProduction para definir qual produção será usada no momento pelo analisador sintático.

A classe genérica TSemantico utiliza-se de ações semânticas para efetuar a geração de código para programas escritos conforme a gramática apresentada no Apêndice A, gerando código similar ao informado pelo usuário. O atributo FFunc guarda uma instância da classe TFunc. Já FProc armazena o código gerado, é que primeiramente processado na ação semântica e armazenado em TStream. O atributo FPreviousAction armazena a ação semântica executada anteriormente.

Através da classe TSemantico, são derivadas as classes que fazem a geração específica para cada SGBD. A classe TGenOracle deriva de TSemantico e sobrescreve os métodos necessários para a geração na linguagem PL/SQL, da mesma forma como a classe TGenSqlServer deriva de TSemantico e irá gerar o código Transact-SQL para o SGBD MS SQL Server. A geração para o SGBD PostgreSQL ocorre da mesma forma, pela derivação da TSemantico. O método sobrescrito é basicamente o ExecuteAction, que processa as ações semânticas através da chamada do método Step, no analisador sintático.

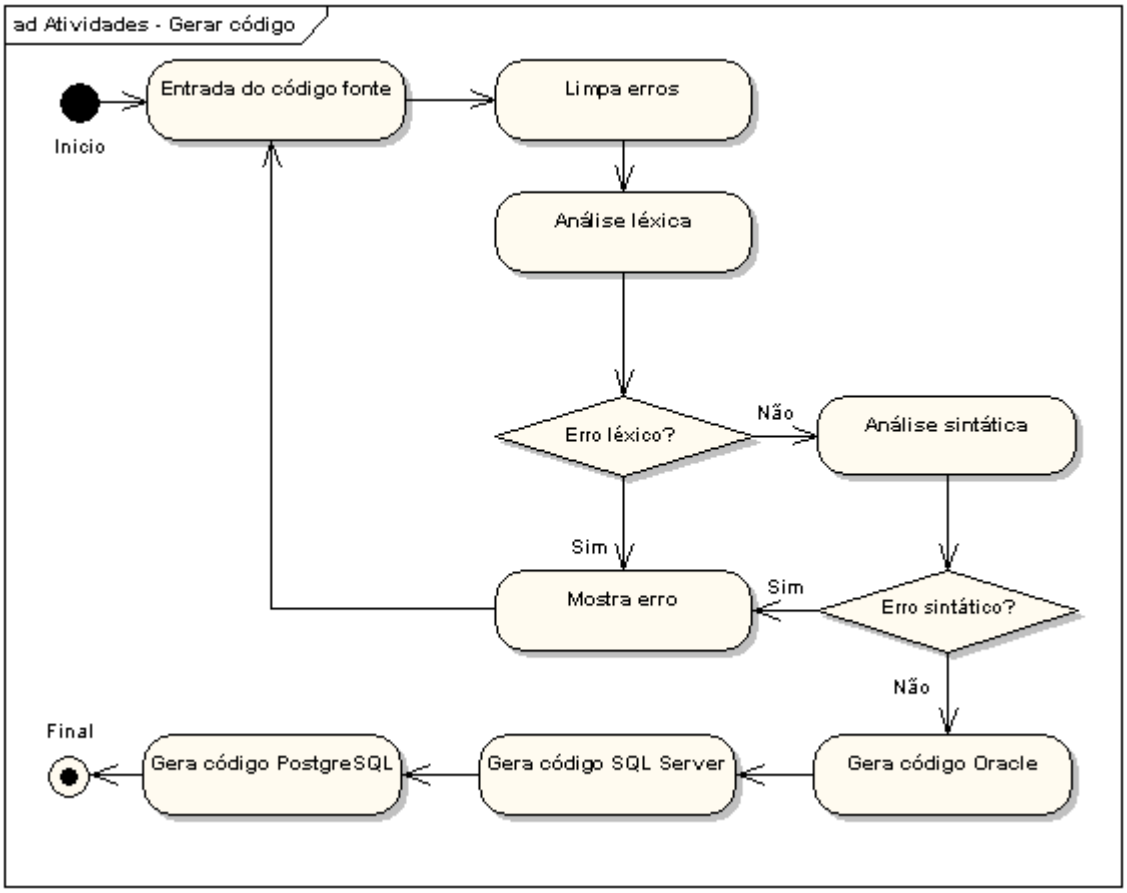
A classe TFunc, possui os atributos para armazenagem de dados referentes a identação. O Atributo FCurrToken representa o *token* que está sendo processado pelo analisador no momento. Já o atributo FLF representa as quebras de linha do código, e FSpace os espaços entre as palavras.

O tipo TToken é representado pela classe TToken. Basicamente, serve para criar um novo *token*, sendo acessada pelos analisadores léxico, sintático e semântico.

Pela classe principal TFMain, os objetos necessários são criados, fazendo as chamadas para compilação do código.

3.2.3 Diagrama de atividades

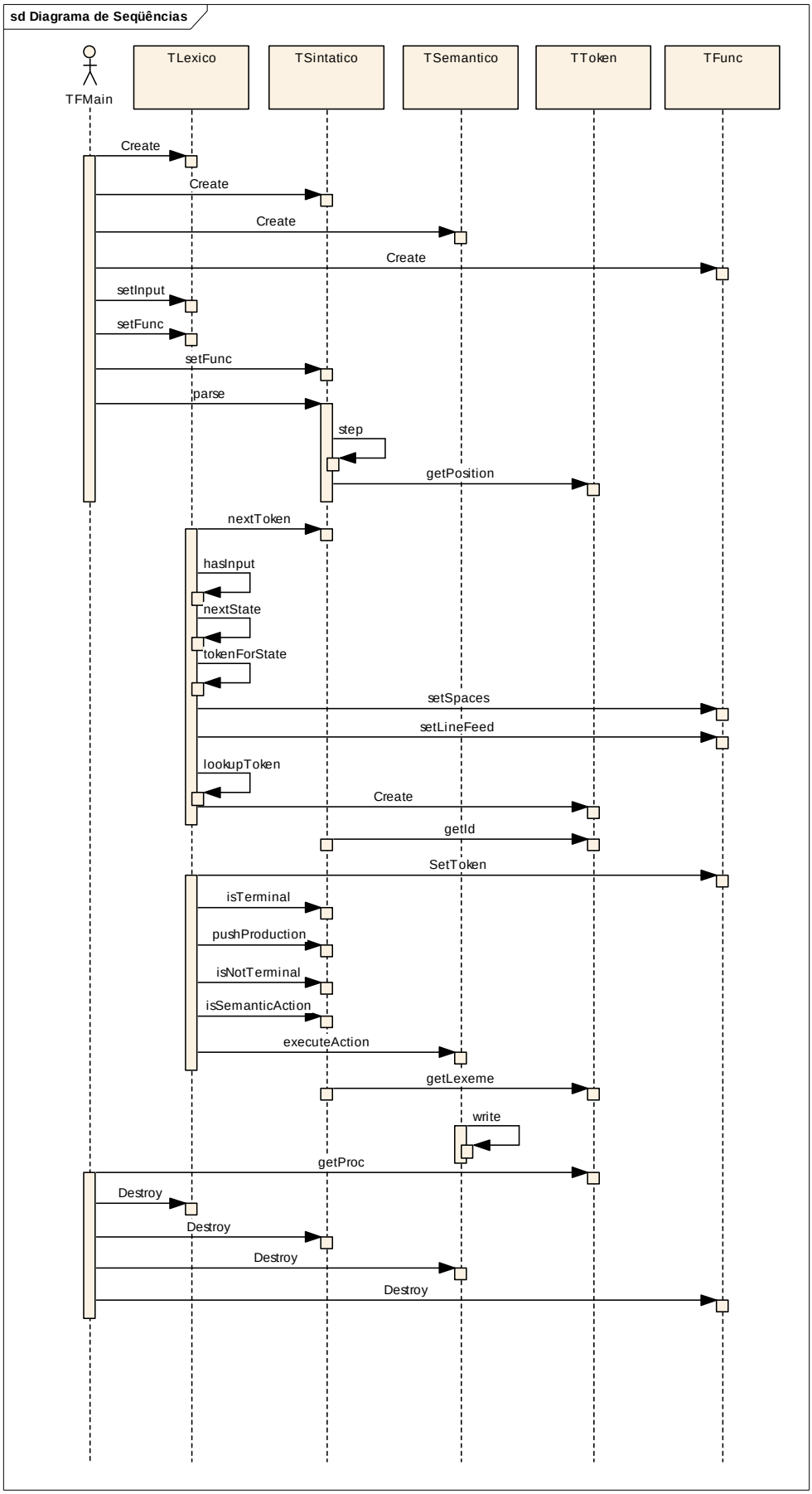
O diagrama de atividades na UML é composto por estados e ações, e oferece uma notação para demonstrar uma seqüência de atividades ou fluxo, como por exemplo, visualizar os passos da execução de um algoritmo (LARMAN, 2001, p 607). No quadro 20 é apresentado o fluxo de atividades para geração de código.



Quadro 20 – Diagrama de atividades da geração de código

3.2.4 Diagrama de Seqüências

Na obra de Larman (2001, p. 118), é afirmado que um diagrama de seqüências é utilizado para representar a iteração entre o usuário e o sistema, para um determinado caso de uso. Através de eventos define-se as mensagens que serão trocadas, sua ordem e os eventos resultantes no sistema. No quadro 21 é exibido o diagrama de seqüências para o caso de uso geração de código.



Quadro 21 – Diagrama de seqüências da geração de código

3.2.5 Autômato finito

O autômato finito determinístico é representado pelas definições regulares apresentadas no quadro 22.

letras_mi: [a-z]	Representa todas as letras minúsculas.
letras_ma: [A-Z]	Representa todas as letras maiúsculas.
exceto_1: [^']	Ignora aspa simples.
exceto_2: [^\n\r]	Ignora *Line Feed* (\n) e Carriage Return (\r).
td_ma : ({letras_ma})	Todas maiúsculas.
td_mi : ({letras_mi})	Todas minúsculas.
traco: ["-"]	Representação de um traço.
digito: [0-9]	Representação de um dígito.
aspa: ["'"]	Representação de uma aspa.
ponto: ["."]	Representação de um ponto.
underscore: ["_"]	Representação do caracter *underscore*.
comment: "/*" [^ "(""*"")"]* "*/"	Comentário de bloco.

Quadro 22 – Definições regulares

```
:[\s\n\r\t]
:!{traco}{traco}{exceto_2}*
:!{comment}
id          : ({td_ma}|{td_mi})({underscore}|{td_ma}|{td_mi}|{digito})*
int         : {digito}({digito})*
literal     : {aspa}{exceto_1}*{aspa}
dec         : {digito}({digito})*{ponto}{digito}({digito})*
```

Quadro 23 – Reconhecimento de *tokens*

No quadro 23, observa-se as regras utilizadas na especificação da seqüência lógica de símbolos que a linguagem assume na formação dos *tokens*. As regras iniciadas com ":" são ignoradas pelo analisador. A primeira delas, :[\s\n\r\t] indica ao analisador para ignorar espaços (\s), quebra de linha (\n), nova linha (\r) e tabulação (\t). A regra :!{traco}{traco}{exceto_2}* indica a leitura de um comentário simples e a regra :!{comment} identifica um comentário de bloco.

As regras id, int, literal e dec serão utilizadas para a leitura de caracteres na definição de um *token*. A regra id representa a formação de um identificador, que pode começar com uma letra maiúscula ou minúscula, seguido de nada ou de uma ou mais letras minúsculas, maiúsculas, dígitos ou *underscores*. Já a regra int define a formação de um inteiro, que

iniciará com um dígito, podendo ser seguido de nenhum ou infinitos dígitos. A regra literal é utilizada para reconhecimento de um literal. Deve iniciar e fechar com aspa, fazendo leitura de qualquer caractere, exceto aspas simples. Por fim, a leitura de um decimal é representada por dec. É definida por um dígito inicial, que pode ser seguido de inúmeros dígitos antes de um ponto, e em seguida terá no mínimo um ou inúmeros dígitos.

3.2.6 Analisador sintático

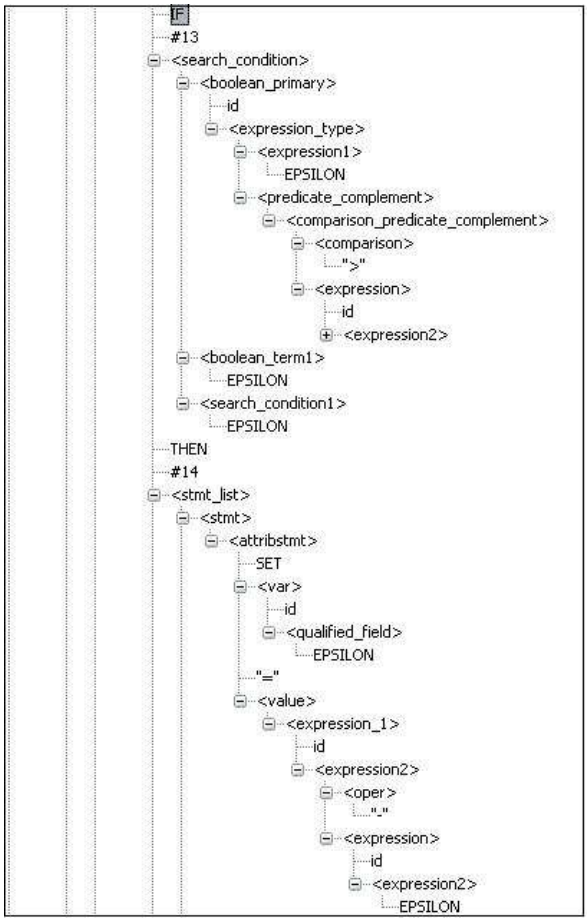
A especificação do analisador sintático foi definida através da utilização de uma GLC, representada utilizando a notação BNF, que pode ser visualizada no Apêndice A.

3.3 IMPLEMENTAÇÃO

Neste capítulo são apresentados detalhes acerca da implementação, com quadros demonstrando parte do código fonte da ferramenta, acompanhado de comentários. Também serão abordadas dificuldades encontradas durante a codificação.

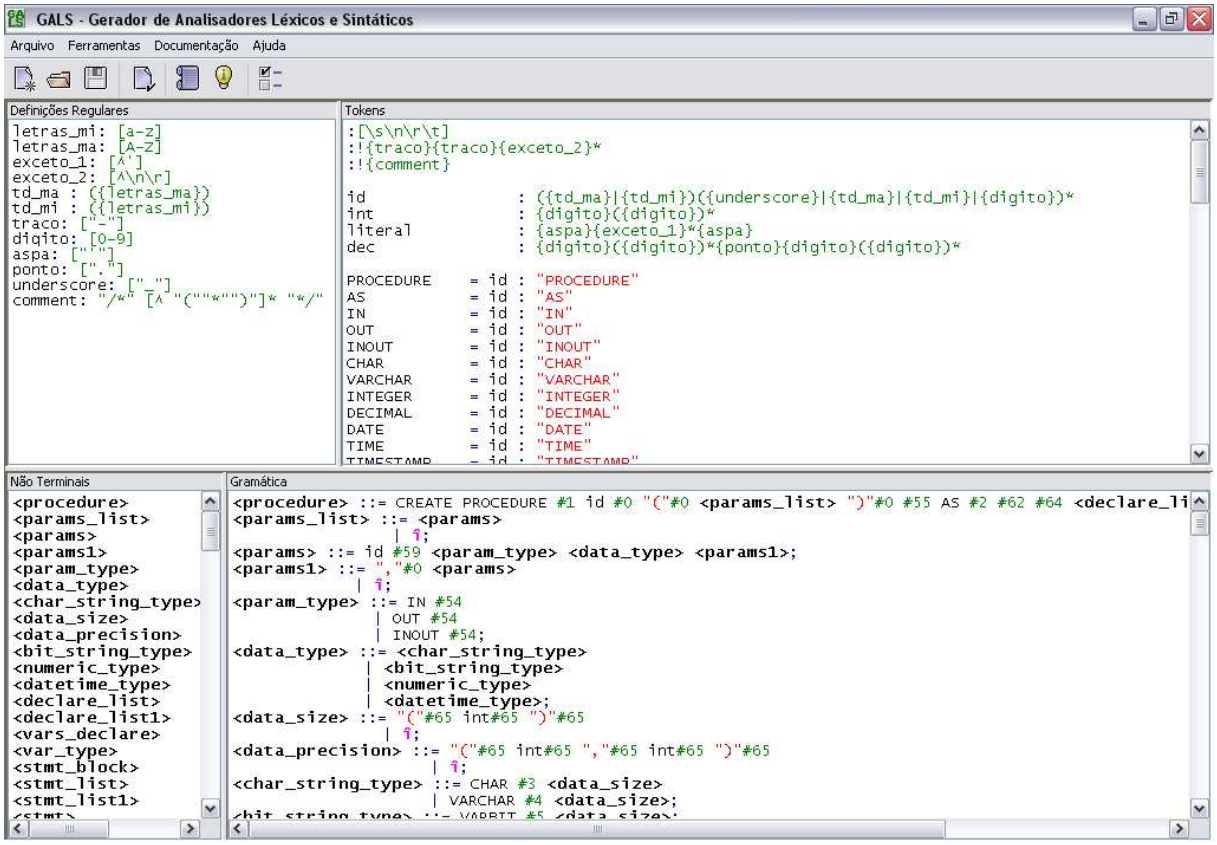
3.3.1 Técnicas e ferramentas utilizadas

Na implementação da ferramenta, fez-se uso do ambiente de desenvolvimento Delphi 7. Para extensão da BNF utilizou-se o gerador GALS versão 2003.10.03 (GESSER, 2003), que implementa através de definições regulares, a sintaxe e a definição dos *tokens* da linguagem em questão, bem como suas regras de produção, com o uso da notação BNF. No quadro 24 pode-se observar a derivação da gramática do comando IF-THEN, através da análise LL(1), utilizando o GALS.



Quadro 24 – Derivação do comando IF-THEN na análise LL(1)

Já o quadro 25 demonstra a tela principal do GALS, onde são escritas as definições regulares, as regras para reconhecimento dos *tokens*, os símbolos não terminais e a gramática da linguagem procedural.



Quadro 25 – Tela principal do GALS

Como o GALS possibilita o desenvolvimento do analisador léxico e sintático, os mesmos foram gerados através dele. Já o analisador semântico criado pelo GALS possui apenas métodos básicos, sendo que a parte de semântica deve ser toda implementada pelo usuário, na forma de ações semânticas, que serão executadas pelo analisador sintático, toda vez que encontrar uma na tabela de *parser*.

Dessa forma, definiu-se na BNF a ação semântica #0, que efetua a cópia dos caracteres lidos pelo analisador. Para as palavras, comandos, cláusulas e funções que divergem em sua definição de um SGBD para o outro, foram criadas ações semânticas, distintas. Como a classe TSemantico promove a herança para as classes que irão gerar o código, há a sobreposição do método ExecuteAction para se adaptar as necessidades do respectivo SGBD. Abaixo, no quadro 26 pode-se verificar o *header* da classe TGenOracle, onde ocorre a sobreposição do método e a derivação da super-classe TSemantico.

```
unit UGenOracle;

interface

uses USemantico, UToken, USemanticError, Dialogs, Classes, UFunc, SysUtils, UConstants;

type
    TGenOracle = class(TSemantico)
    public
      procedure executeAction(action : integer; const token : TToken); override;
    end;
```

Quadro 26 – *Header* da classe TGenOracle

Na execução de uma ação semântica, o *token* que será disponibilizado para a ação é o último lido antes da chamada da mesma. A geração de código é feita substituindo os comandos da gramática original pelos comandos do respectivo SGBD, conforme demonstra o quadro 27. A inserção de caracteres e a manipulação na ordem dos *tokens* que formarão os comandos é toda feita por ações semânticas, já que oferecem com precisão a localização atual do cursor na leitura do código a ser traduzido.

```
//#44 - substr
44: Write('SUBSTRING');
//#45 - length
45: Write(cTemp);
//#46 - pos
46: Write('POSITION');
//#47 - get_datetime
47: Write('CURRENT_TIMESTAMP');
//#48 - set
48: ;
//#49 - "="
49: Write(':=');
//#50 - select
50: Write(cTemp);
//#51 - in
51: Write('IN');
```

Quadro 27 – Ações semânticas

O método *Write*, exibido no quadro 28, é chamado por grande parte das ações semânticas, salvo as que não são necessárias, como a ação 48 no quadro 27. O mesmo irá fazer as quebras de linha e posicionar os espaços corretamente entre as palavras. Também armazenará em o *buffer* que contém o comando que está sendo montado.

```
procedure TSemantico.Write(Cmd : String);
begin
    //se quebra de linha
    if Cmd = '#$D#$A' then begin
      //verifica se não é uma linha perdida, ou um ";"
      if (Trim(FStream) <> '') and (Trim(FStream) <> ';') then
        //adiciona na lista
        FProc.Add(FStream);
      //identação
      FStream := FIdent;
    end else begin
      //se ação anterior for a #53, tira espaços a direita e coloca o ";"
      if Cmd = ';' then
        FStream := TrimRight(FStream) + Cmd
      else
        if Copy(Cmd,1,1) = '%' then
          FStream := TrimRight(FStream) + Cmd + Copy(FFunc.GetSpaces,1,1)
        else
          FStream := FStream + Cmd + Copy(FFunc.GetSpaces,1,1);
    end;
    //limpa espaços.. prá não estragar o espaçamento
    FFunc.ClearSpaces;
end;

end.
```

Quadro 28 – Método *Write*

A conexão com os SGDBs abordados foi realizada com a suíte de componentes SQLDirect 4.2.0 (SHEINO, 2005), que possibilita o acesso nativo aos mesmos. Como a parte onde o código gerado vai ficar disponível para o usuário foi implementada em um TFrame, ocorreram erros de acesso a memória caso um componente TSDDataBase ou um TSDQuery fosse colocado no mesmo, conflitando com as instâncias do TFrame. Dessa forma, optou-se por criar dinamicamente os componentes, evitando assim o erro.

O desenvolvimento das novas estruturas objetivando a extensão da gramática original deu-se através de detalhes da construção da gramática apresentada na BNF ISO/IEC 9075-2:2003 (SQL/FOUNDATION, 2005), que apresenta as mais recentes construções sintáticas para a linguagem SQL.

Os detalhes de geração de código para o SGDB Oracle foram obtidos em TECH ON THE NET (2006). Para o SGBD MS SQL Server as informações foram encontradas em Microsoft Corporation (2006). Já para o SGBD PostgreSQL, o detalhes foram extraídos de POSTGRESQL (2006).

3.3.2 Operacionalidade da implementação

Ao iniciar a ferramenta, a tela principal, apresentada na figura 11, será exibida ao usuário. Através dela pode-se entrar com o código fonte digitando o mesmo ou fazendo leitura de um arquivo já salvo. A tela principal também possibilita ao usuário fazer a compilação do código fonte e fazer a manipulação dos dados digitados, com os botões Copiar, Colar, Excluir e Desfazer.

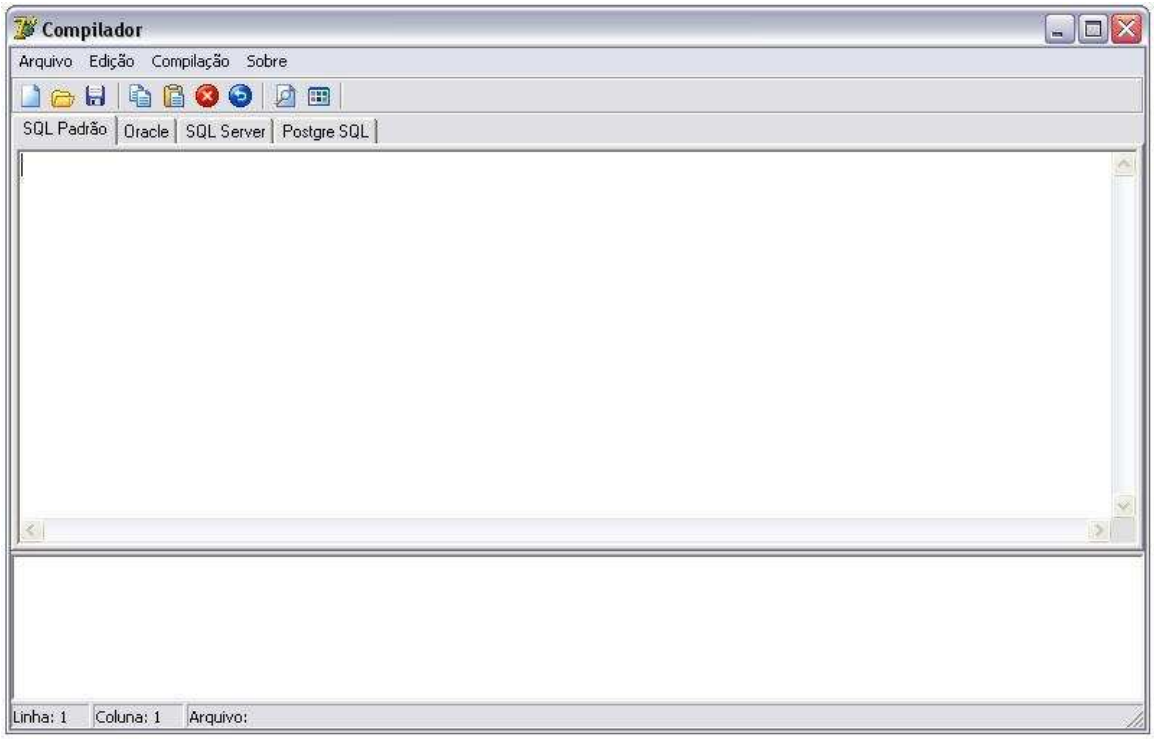


Figura 11 – Tela principal

A tela exibida na figura 12 demonstra o tratamento de erros feito pela compilação. Ao se deparar com um erro, a ferramenta deve exibir o mesmo ao usuário de forma clara e concisa.

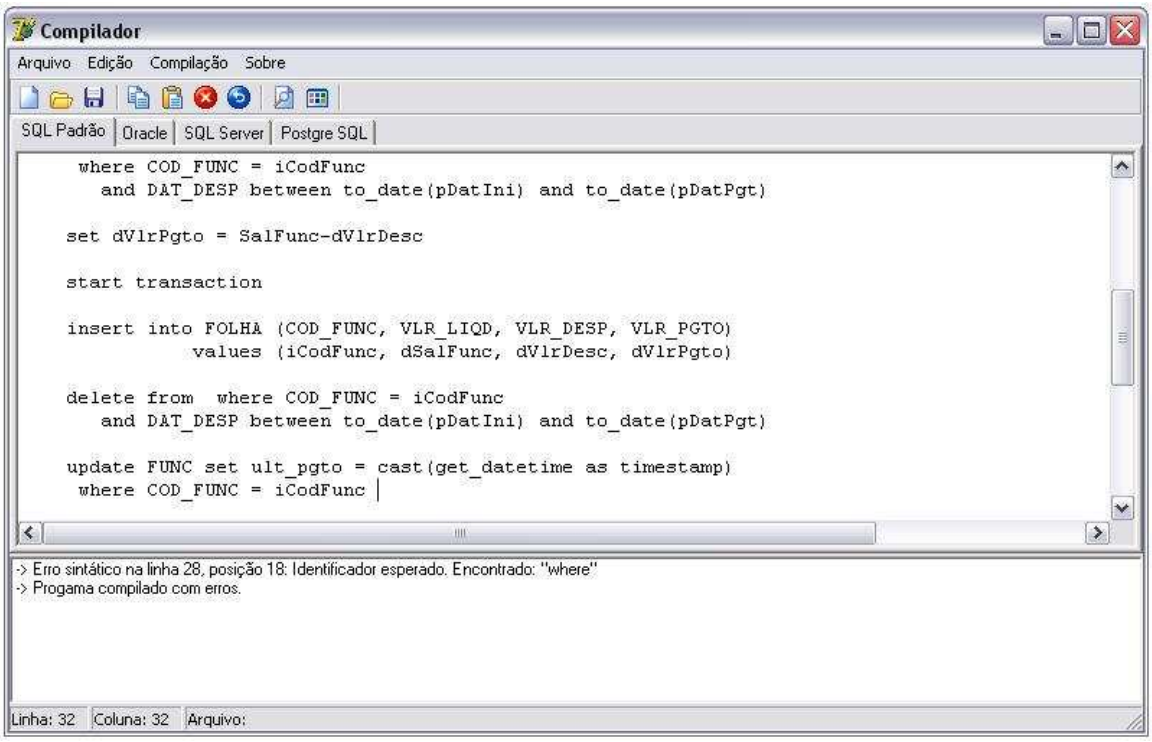


Figura 12 – Verificação de erros

Na figura 13 observa-se a tela de conexão com o banco de dados, onde o usuário insere informações referentes ao Servidor, Usuário e Senha para efetuar a conexão. Já na figura 14 é exibida a mensagem de retorno do SGDB, ao efetuar a criação de um procedimento, que no exemplo, já existe no SGDB.

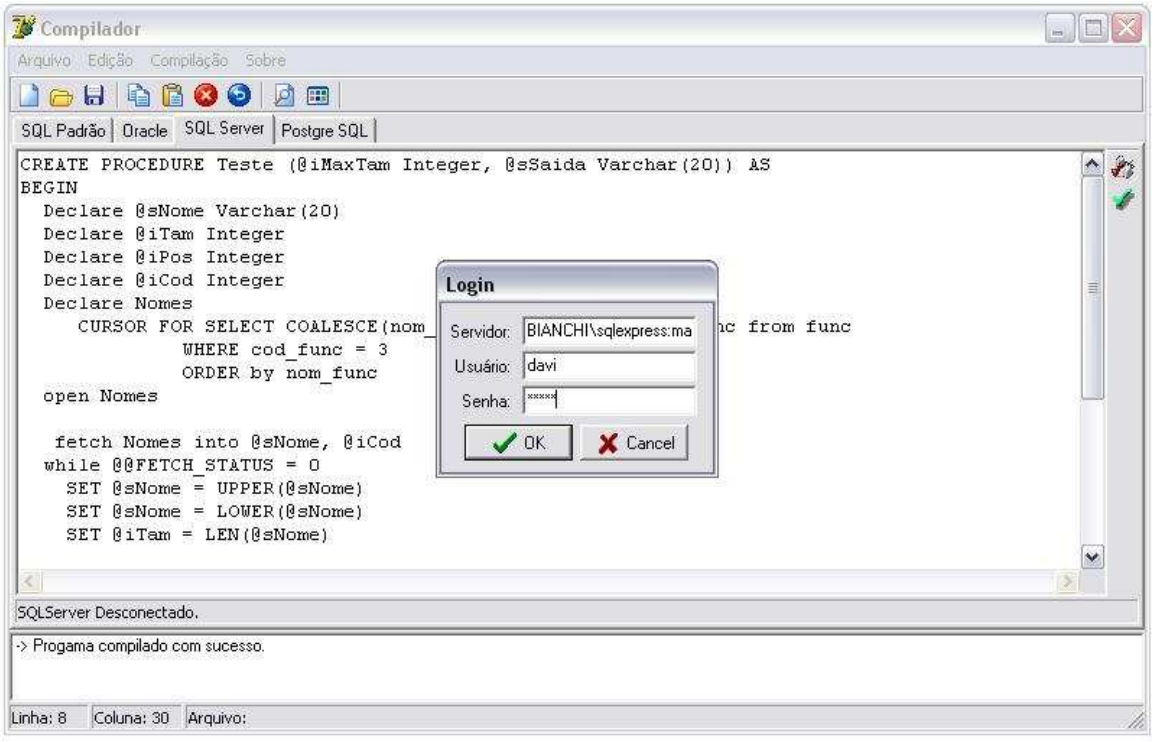


Figura 13 – Conexão com banco de dados

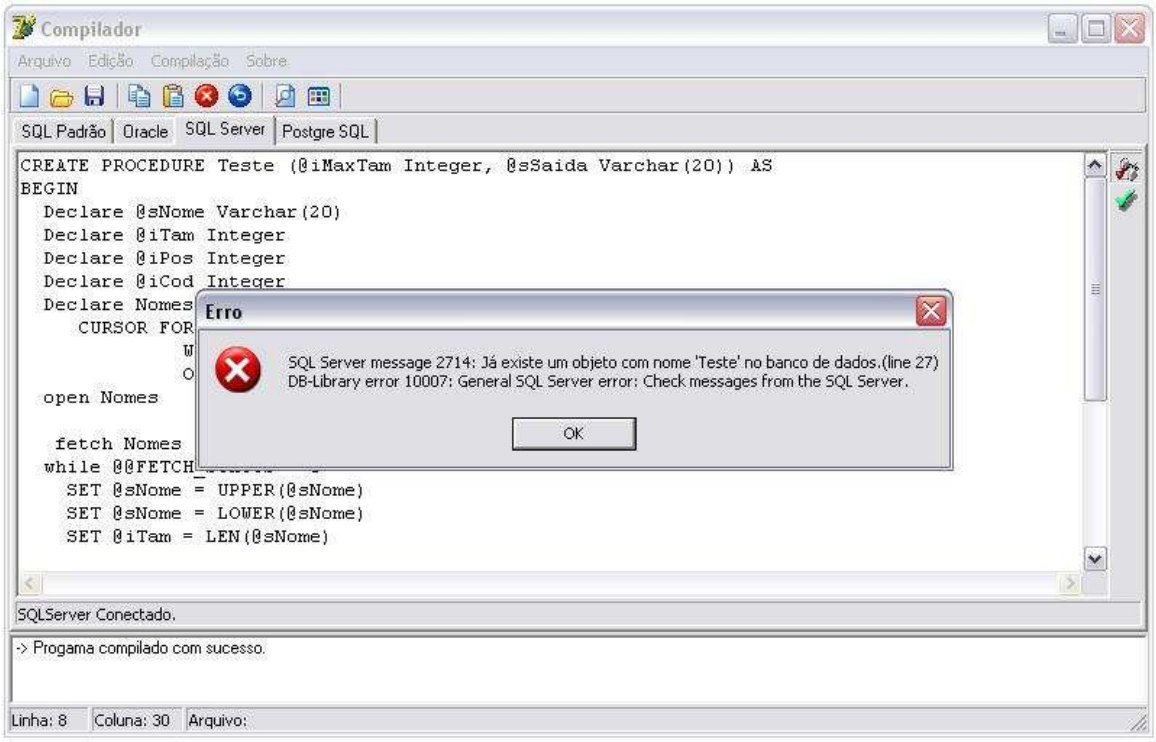


Figura 14 – Criação do procedimento

No quadro 29 é exibido o código fonte gerado para o SGBD Oracle exemplo usado nas figuras 12 e 13.

```
CREATE OR REPLACE PROCEDURE Teste (iMaxTam IN NUMBER, sSaida OUT VARCHAR2) IS
  sNome VARCHAR2(20);
  iTam NUMBER;
  iPos NUMBER;
  iCod NUMBER;
  CURSOR
    Nomes IS SELECT COALESCE(nom_func,'Sem Nome'), cod_func FROM func
         WHERE cod_func = 3
      ORDER BY nom_func;
BEGIN
  OPEN Nomes;
  LOOP
    FETCH Nomes INTO sNome, iCod;
    EXIT WHEN Nomes%NOTFOUND;
    sNome := UPPER(sNome);
    sNome := LOWER(sNome);
    iTam := LENGTH(sNome);
    iPos := INSTR('a',sNome);
  IF iTam > 20 THEN
    sSaida := SUBSTR(sNome,1,20);
  ELSE
    sSaida := sNome;
  END IF;
  UPDATE func SET nom_func = sSaida WHERE cod_func = iCod;
  COMMIT;
END LOOP;
CLOSE Nomes;
END;
```

Quadro 29 – Fonte gerado para o SGBD Oracle

No quadro 30 é exibido o código fonte gerado para o SGBD MS SQL Server exemplo usado nas figuras 12 e 13.

```
CREATE PROCEDURE Teste (@iMaxTam INTEGER, @sSaida VARCHAR(20)) AS
BEGIN
  DECLARE @sNome VARCHAR(20)
  DECLARE @iTam INTEGER
  DECLARE @iPos INTEGER
  DECLARE @iCod INTEGER
  DECLARE Nomes
     CURSOR FOR SELECT COALESCE(nom_func,'Sem Nome'), cod_func FROM func
          WHERE cod_func = 3
        ORDER by nom_func
OPEN Nomes
   FETCH Nomes INTO @sNome, @iCod
WHILE @@FETCH_STATUS = 0
  SET @sNome = UPPER(@sNome)
  SET @sNome = LOWER(@sNome)
  SET @iTam = LEN(@sNome)
  SET @iPos = CHARINDEX('a',@sNome)
  IF @iTam > 20
    SET @sSaida = SUBSTRING(@sNome,1,20)
  ELSE
    SET @sSaida = @sNome
  BEGIN TRANSACTION
  UPDATE func SET nom_func = @sSaida WHERE cod_func = @iCod
  COMMIT
 CLOSE Nomes
 DEALLOCATE Nomes
END
```

Quadro 30 – Fonte gerado para o SGBD MS SQL Server

No quadro 31 é exibido o código fonte gerado para o SGBD PostgreSQL exemplo usado nas figuras 12 e 13.

```
CREATE OR REPLACE FUNCTION Teste (iMaxTam IN NUMERIC, sSaida OUT VARCHAR(20)) RETURNS VARCHAR(20) AS $$
DECLARE
  sNome VARCHAR;
  iTam NUMERIC;
  iPos NUMERIC;
  iCod NUMERIC;
 Nomes CURSOR FOR
     SELECT COALESCE(nom_func,'Sem Nome'), cod_func FROM func
          WHERE cod_func = 3
        ORDER BY nom_func;
BEGIN
  OPEN Nomes;
  LOOP
    FETCH Nomes INTO sNome, iCod;
    EXIT WHEN Nomes%NOTFOUND;
    sNome := UPPER(sNome);
    sNome := LOWER(sNome);
    iTam := LENGTH(sNome);
    iPos := POSITION('a' IN sNome);
    IF iTam > 20 THEN
      sSaida := SUBSTRING(sNome,1,20);
    ELSE
      sSaida := sNome;
    END IF;
    UPDATE func SET nom_func = sSaida WHERE cod_func = iCod;
   COMMIT;
 END LOOP;
  CLOSE Nomes;
END;
$$ LANGUAGE PLpgSQL;
```

Quadro 31 – Fonte gerado para o SGBD PostgreSQL

3.4 RESULTADOS E DISCUSSÃO

Primeiramente, a extensão se deu por uso do código implementado por Hiebert (2003), devido ao fato de possuir grande parte das estruturas que forneceriam uma base sólida para as novas customizações. Entretanto, após o inicio do desenvolvimento de estruturas para contemplar as novas funções propostas neste trabalho, verificou-se alguns erros na BNF da linguagem, impactando na definição já existente das estruturas de dados.

Uma vez que as correções não apresentaram resultados satisfatórios, optou-se por utilizar a ferramenta GALS (GESSER, 2003), eliminando dessa forma fatorações e ambigüidades presentes na BNF, e reduzindo, conseqüentemente, problemas de compilação e geração de código.

A utilização da ferramenta GALS (GESSER, 2003) para definição e testes da BNF e para geração do analisador léxico e sintático facilitaram em grande parte o desenvolvimento deste trabalho, pois através da criação de ações semânticas na BNF, o caminho para a geração de código utilizando as novas estruturas propostas se tornou mais simples e funcional.

Como característica negativa, o modelo de analisador definido pelo GALS (GESSER, 2003) ao encontrar um erro aponta o mesmo de forma coerente, porém, o processo de compilação é encerrado, não identificando erros que possam estar presentes após esse primeiro erro.

Outra limitação é que todas as variáveis devem ser declaradas no início do procedimento, por ser limitação sintática de alguns SGBDS, e expressões lógicas não devem apresentar parênteses, evitando dessa forma redundância na gramática.

4 CONCLUSÕES

A idéia de extensão do trabalho desenvolvido por Hiebert (2003) se apresentou como uma alternativa viável e interessante, pois a praticidade de se escrever um código fonte somente uma vez e conseguir gerar código para diferentes SGBDs é de um significado imenso para os que lidam com esse trabalho no cotidiano.

Durante o desenvolvimento muitas novas idéias foram adotadas, visando ir além da meta inicial, porém algumas foram descartadas pela proporção imensa que isso daria ao trabalho, e que talvez não possibilitassem a conclusão do mesmo em tempo hábil. Essas idéias ficarão definidas como extensões, juntamente com as propostas no trabalho desenvolvido por Hiebert (2003).

Neste trabalho, o autor explora as extensões do trabalho de Hiebert (2003), porém, fazendo uso de um gerador para os analisadores léxico e sintático, visando uma melhora na definição da linguagem, eliminando fatorações e ambigüidades encontradas na definição original. O objetivo disso é um aumento na qualidade da compilação e da geração de código, bem como facilitar futuros trabalhos que possam fazer correlação com esse, através das extensões apontadas no capítulo de conclusões.

O autor ainda implementa uma funcionalidade não apontada por Hiebert (2003) nas extensões de seu trabalho, que é a opção para realizar a conexão com os SGBDs abordados na geração de código, possibilitando assim a execução do código gerado sem uso de outra ferramenta, fazendo dessa forma a validação semântica do procedimento desenvolvido.

Observando a conclusão do trabalho, verifica-se que seus objetivos propostos foram atingidos, bem como objetivos não definidos inicialmente, como a conexão com os SGBDs, foram incorporados e apresentaram ótima funcionalidade.

Vale ressaltar que um ponto negativo do mesmo é a impossibilidade de utilização de recursos específicos de algum SGBD abordado para uso em qualquer outro também abordado. Se tratando de uma geração de código em grande parte genérica, pode-se esperar que nem sempre o código gerado terá a performance mais adequada e utilizará ao máximo os recursos fornecidos pelo SGBD.

4.1 EXTENSÕES

Nesta seção são apresentadas sugestões de extensão e adaptação para este trabalho, assim como extensões definidas por Hiebert (2003) em seu trabalho de conclusão que não foram exploradas.

Como possíveis extensões, enumera-se:

a) possibilitar a geração de código para outros SGBDs;
b) definir novas funções para a linguagem estendida nesse trabalho. Como exemplo, pode-se citar funções de agregação, funções de *string*, numéricas ou de datas ainda não abordadas, estruturas "*case*" e "*join*";
c) fazer a engenharia reversa para análise e/ou comparação de tipos, e verificação de parâmetros para chamadas de outras *procedures*;
d) implementar um depurador para essa linguagem;
e) introduzir a utilização de *templates* para geração de código;
f) migração da ferramenta para uma tecnologia mais atual, fornecendo suporte para utilização da mesma remotamente.

REFERÊNCIAS BIBLIOGRÁFICAS

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

GESCHWINDE, E.; SCHÖNIG, H. **PostgreSQL**: developers handbook. Indianapolis: Sams, 2002.

GESSER, C. E. **GALS:** gerador de analisadores léxicos e sintáticos. [S.l.], [2003]. Disponível em: <http://gals.sourceforge.net/>. Acesso em: 06 nov. 2006.

GROFF, J. R.; WEINBERG, P. N. **SQL**: the complete reference. Berkeley: Osborne/McGraw-Hill, 1999.

GRUNE, D. et al. **Projeto moderno de compiladores**: implementação e aplicações. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2001.

HIEBERT, D. **Protótipo de um compilador para a linguagem PL/SQL**. 2003. 52 f. Trabalho de Conclusão do Curso (Bacharelado em Ciência da Computação) – Centro de Ciências Exatas e Naturais, Universidade Regional de Blumenau, Blumenau.

HOPCROFT, J.E.; ULLMAN, J. D.; MOTWANI, R. **Introdução a teoria dos autômatos, linguagens e computação**, 2nd ed. Tradução Vandenberg D. de Souza. Rio de Janeiro: Campus, 2002.

JOSÉ NETO, J. **Introdução à compilação**. Rio de Janeiro: LTC, 1987.

KRIEGEL, A.; TRUKHNOV B. M. **SQL bible**. Indianapolis: Wiley, 2003.

LAKSHMAN, B. **Oracle 9i PL/SQL**: a developer's guide. Berkeley: Apress, 2003.

LARMAN, C. **Applying UML and Patterns**: an introduction to object-oriented analysis and design and the unified process, 2. ed. Indianapolis: Prentice Hall Ptr, 2001.

MICROSOFT CORPORATION. **SQL Server Express**. [S.l.], [2006]. Disponível em: <http://msdn.microsoft.com/vstudio/express/sql/>. Acesso em: 10 nov. 2006.

OLIVEIRA, Wilson José. **Oracle 8i & PL/SQL**. Florianópolis: Visual Books, 2000.

PETKOVIC, D. **SQL Server 2000:** guia prático. Tradução César Camargos, João Tortello, Rogério Maximiliano. São Paulo: Makron Books, 2001.

POSTGRESQL. **PostgreSQL 8.1.5 documentation**. [S.l.], [2006]. Disponível em: <http://www.postgresql.org/docs/8.1/interactive/index.html>. Acesso em: 10 nov. 2006.

PRICE, A. M. A.; TOSCANI, S. S. **Implementação de linguagens de programação**: compiladores. 2. ed. Porto Alegre: Sagra Luzzatto, 2001.

RAMAKRISHNAN, R.; GEHRKE, J. **Database management systems**. 2nd ed. New York: McGraw-Hill, 1999.

SHEINO, Y. **SQLDirect library**. [S.l.], [2006]. Disponível em: <http://www.sqldirect-soft.com/>. Acesso em: 10 nov. 2006.

SPARX SYSTEMS. **Enterprise architect:** UML design tools. [S.l.], [2006]. Disponível em: <http://www.sparxsystems.com/products/ea.html>. Acesso em: 04 out. 2006.

SQL/FOUNDATION. **BNF grammar for ISO/IEC 9075-2:2003**: database language SQL (SQL-2003). [S.l.], [2005]. Disponível em: <http://savage.net.au/SQL/sql-2003-2.bnf.html>. Acesso em: 06 out. 2006.

SUNDERIC, D.; WOODHEAD, T. **SQL Server 2000**: stored procedure programming. Berkeley: Osborne/McGraw-Hill, 2000.

TAYLOR, A. G. **SQL for dummies.** 5th ed. Indianapolis: Wiley, 2003.

TECH ON THE NET. **Oracle/PLSQL topics:** built-in functions. [S.l.], [2006]. Disponível em: <http://www.techonthenet.com/oracle/functions/>. Acesso em: 23 out. 2006.

SQL In: WIKIPEDIA, a enciclopédia livre. [S.l.]: Wikimedia Foundation, 2006. Disponível em: <http://en.wikipedia.org/wiki/SQL>. Acesso em: 03 out. 2006.

APÊNDICE A – GRAMÁTICA DA LINGUAGEM DE BANCO DE DADOS

O reconhecimento dos *tokens* pertencentes à linguagem utilizada para criação dos procedimentos armazenados dá-se através dos elementos básicos descritos abaixo, conforme definição realizada através de autômatos finitos:

a) id = letra (letra | "_" | digito)*

b) int = digito(digito)*

c) dec = digito(digito)*"."digito(digito)*

d) literal = """(qualquer_caracter_menos_aspa)*"""

As regras sintáticas para construção de sentenças adequadas à gramática da linguagem estão descritas no quadro 32, utilizando a notação BNF.

```
<procedure> ::= CREATE PROCEDURE id ( <params_list> ) AS
<declare_list> <stmt_block>
<params_list> ::= <params>
                | ε
<params> ::= id <param_type> <data_type> <params1>
<params1> ::= , <params>
            | ε
<param_type> ::= IN
               | OUT
               | INOUT
<data_type> ::= <char_string_type>
               | <bit_string_type>
               | <numeric_type>
               | <datetime_type>
<data_size> ::= ( int )
               | ε
<data_precision> ::= ( int , int )
                   | ε
<char_string_type> ::= CHAR <data_size>
                     | VARCHAR <data_size>
<bit_string_type> ::= VARBIT <data_size>
```

```
<numeric_type> ::= INTEGER
                 | DECIMAL <data_precision>
<datetime_type> ::= DATE
                  | TIME
                  | TIMESTAMP
<declare_list> ::= DECLARE <declare_list1>
                  | ε
<declare_list1> ::= <vars_declare>
                  | <cursor>
<vars_declare> ::= id <var_type> <declare_list>
<var_type> ::= <data_type>
<stmt_block> ::= BEGIN <stmt_list> <tryexcept> END
<stmt_list> ::= <stmt> <stmt_list1>
<stmt_list1> ::= <stmt_list>
                | ε
<stmt> ::= <dmlstmt>
         | <attribstmt>
         | <ifstmt>
         | <whilestmt>
         | <raisestmt>
         | <cursor_stmt>
         | <loop_stmt>
         | <exit_when>
         | START TRANSACTION
         | ROLLBACK
         | COMMIT
<dmlstmt> ::= <select>
            | <insert>
            | <update>
            | <delete>
<type_func> ::= <char_function>
              | <conversion_function>
              | <sysdate_function>
              | <count_function>
```

```
              | <coalesce_function>
<select> ::= SELECT <select_columns> <into> FROM <table_list>
<search> <order_by> <group_by>
<select_columns> ::= *
                   | <sel_columns_list>
<sel_columns_list> ::= id <qualified_field><sel_columns_list1>
                     | <type_func> <sel_columns_list1>
<sel_columns_list1> ::= , <sel_columns_list>
                       | ε
<sel_columns_list2> ::= <type_func> <sel_columns_list3>
                       | id <sel_columns_list3>
                       | <prim_type> <sel_columns_list3>
<sel_columns_list3> ::= , <sel_columns_list2>
                       | ε
<prim_type_or_null> ::= <prim_type>
                       | NULL
<qualified_field> ::= . id
                     | ε
<count_function> ::= COUNT( <select_columns> )
<into> ::= INTO <symbols_list>
         | ε
<symbols_list> ::= id <symbols_list1>
<symbols_list1> ::= , <symbols_list>
                   | ε
<table_list> ::= id <alias> <table_list1>
<table_list1> ::= , <table_list>
                 | ε
<alias> ::= id
          | ε
<search> ::= WHERE <search_condition>
           | ε
<search_condition>   ::=   <boolean_primary>   <boolean_term1>
<search_condition1>
| NOT <boolean_primary> <boolean_term1> <search_condition1>
```

```
<search_condition1> ::= OR <search_condition>
| ε
<boolean_term> ::= <boolean_primary> <boolean_term1>
                 | NOT <boolean_primary> <boolean_term1>
<boolean_term1> ::= AND <boolean_term>
                   | ε
<boolean_factor> ::= <boolean_primary>
                   | NOT <boolean_primary>
<boolean_primary> ::= EXISTS ( <select> )
                    | <prim_type> <expression1>
<predicate_complement>
                   | ( <predicate_type>
                   | id <expression_type>

<predicate_type>::= id <predicate_type_1>
                     | <prim_type> <expression1> <prim_type_1>
                     |  EXISTS  (  <select>  )  <boolean_term1>
<search_condition1> )
                     |   (   <predicate_type>   <boolean_term1>
<search_condition1>  )
                     |  NOT  <boolean_primary>  <boolean_term1>
<search_condition1> )
<predicate_type_1>::= <expression1> <predicate_type_2>
                     |      <cursor_state>      <boolean_term1>
<search_condition1>  )
<predicate_type_2>::=  <predicate_complement>  <boolean_term1>
<search_condition1>  )
                     | ) <predicate_complement>
<prim_type_1>::=  ) <predicate_complement>
                     |   <predicate_complement>   <boolean_term1>
<search_condition1>  )
<expression_type>::= <expression1> <predicate_complement>
                   | <cursor_state>
<expression> ::= id <expression2>
```

```
                |  <prim_type> <expression1>
                | ( <term> <expression1> )
                | <type_func> <expression1>
<expression_1> ::= id <expression2>
                  |  <prim_type> <expression2>
                  | ( <term> <expression2> )
                  | <type_func> <expression2>
<oper> ::= + | - | / | *
<comparison> ::=  = | < | <= | > | >= | <>
<expression1> ::= <oper> <expression>
                  | ε
<expression2> ::= <oper> <expression>
                  | , <expression_1>
                  | "=" <value>
                  | ε
<prim_type> ::= literal
               | int
               | dec
<term> ::= id
          | <prim_type>
<predicate_complement> ::= <between_predicate_complement>
                          | <comparison_predicate_complement>
                          | <null_predicate_complement>
<between_predicate_complement>  ::=  BETWEEN  <expression>  AND
<expression>
<comparison_predicate_complement> ::= <comparison><expression>
<null_predicate_complement> ::= IS <nullable>
<nullable> ::= NULL
              | NOT NULL
<insert>    ::=    INSERT    INTO    id    <insert_columns_list>
<insert_values>
<insert_columns_list> ::= ( <columns_list> )
                         | ε
<columns_list> ::= id <columns_list1>
```

```
<columns_list1> ::= , <columns_list>
                  | ε
<insert_values> ::= <values>
                      | <select>
<values> ::= VALUES <values_list>
<values_list> ::= <value> <values_list1>
<values_list1> ::= , <values_list>
                 | ε
<value> ::= NULL
          | <expression_1>
<update> ::= UPDATE id SET <update_col_value_list> <search>
<update_col_value_list>        ::=        <update_column_value>
<update_col_value_list1>
<update_col_value_list1> ::= , <update_col_value_list>
                           | ε
<update_column_value> ::= id = <value>
<delete> ::= DELETE FROM id <search>
<attribstmt> ::= SET <var> = <value>
<var> ::= id <qualified_field>
<ifstmt> ::= IF <search_condition> THEN <new_stmt> <else>
<new_stmt> ::= <stmt>
             | <stmt_block>
<else> ::= ELSE <new_stmt>
         | ε
<whilestmt> ::= WHILE <search_condition> DO <new_stmt>
<raisestmt> ::= RAISEERROR literal
<tryexcept> ::= EXCEPT <exceptions_list>
              | ε
<exceptions_list> ::= <exception> <exceptions_list1>
<exceptions_list1> ::= <exceptions_list>
                     | ε
<exception> ::= WHEN id THEN <except>
<except> ::= <new_stmt>
           | NULL
```

```
<order_by> ::= ORDER BY <sel_columns_list> <order_by_ord>
             | ε
<order_by_ord> ::= ASC
                 | DESC
                 | ε
<group_by> ::= GROUP BY <sel_columns_list> <group_by_having>
             | ε
<group_by_having> ::= HAVING ( <search_condition> )
                    | ε
<cursor> ::= CURSOR id IS <select> <cursor_for_upd>
<cursor_for_upd> ::= FOR UPDATE
                   | ε
<cursor_stmt> ::= OPEN id
                | CLOSE id
                | FETCH id <into>
<cursor_state> ::= FOUND
                 | NOTFOUND
                 | ISOPEN
<loop_stmt> ::= LOOP <stmt_list> END LOOP
<exit_when> ::= EXIT WHEN <search_condition>
<coalesce_function> ::= COALESCE ( <sel_columns_list2> )
<conversion_function> ::= <to_char_function>
                        | <to_date_function>
                        | <to_number_function>
                        | <cast_function>
<format_mask> ::= , literal
                | ε
<col_or_prim> ::= <sel_columns_list>
                | <prim_type>
<col_or_literal> ::= id
                   | literal
                   | <str_function>
<str_function> ::= <upper_function>
                 | <lower_function>
```

```
                    | <substr_function>
<to_char_function> ::= TO_CHAR ( <col_or_prim> <format_mask> )
<to_date_function> ::= TO_DATE ( <col_or_prim> <format_mask> )
<to_number_function>      ::=      TO_NUMBER(      <col_or_prim>
<format_mask> )
<cast_function> ::= CAST (<col_or_prim> AS <data_type>)
<char_function> ::= <upper_function>
                    | <lower_function>
                    | <substr_function>
                    | <length_function>
                    | <pos_function>
                    | ε
<upper_function> ::= UPPER (<col_or_literal>)
<lower_function> ::= LOWER (<col_or_literal>)
<substr_function> ::= SUBSTR (<col_or_literal>, int , int )
<length_function> ::= LENGTH (<col_or_literal>)
<pos_function> ::= POS ( <col_or_literal> IN <col_or_literal>)
<sysdate_function> ::= GET_DATETIME
```

Quadro 32 – Regras sintáticas da linguagem