UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS DE SÃO CARLOS

**UMA LINGUAGEM PARA O
GERENCIADOR DE OBJETOS BASEADA NO
MODELO DE REPRESENTAÇÃO DE OBJETOS**

PEDRO LUIZ PIZZIGATTI CORRÊA

Orientador: PROF. DR. CAETANO TRAINA JR.

Dissertação apresentada ao Instituto de Ciências Matemáticas de São Carlos, da Universidade de São Paulo, como parte dos requisitos para a obtenção do título de Mestre na Área de Ciências de Computação e Matemática Computacional.

SÃO CARLOS
Estado de São Paulo - Brasil
Maio - 1992

Este trabalho é dedicado ao meu irmão Cláudio.

AGRADECIMENTOS

Ao Prof. Dr. Caetano Traina Jr. orientador deste trabalho, pela amizade, paciência e compreensão.

Ao Núcleo Tecnológico para Informática Agropecuária (NTIA) da Empresa Brasileira de Pesquisa Agropecuária (EMBRAPA), por possibilitar e apoiar o desenvolvimento deste trabalho.

Aos participantes do Grupo de Base de Dados e Engenharia de Software do ICMSC-USP, que contribuíram com muitas idéias deste trabalho.

Ao Instituto de Ciências Matemáticas de São Carlos da Universidade de São Paulo, que vêm proporcionando-me aprimoramento técnico e científico.

Ao Instituto de Física e Química de São Carlos da Universidade de São Paulo, onde iniciei o mestrado.

Ao Conselho Nacional de Pesquisa e Desenvolvimento Científico e Tecnológico (CNPq), que patrocinou este trabalho através de bolsa de mestrado.

Aos meus pais Pedro e Irene, pelo apoio, dedicação e amor que sempre recebi.

À tia Clemência que me alojou em São Carlos, permitindo-me efetuar meus estudos desde a graduação.

Aos meus queridos irmãos Edson Luiz e Maria Elvira, pela amizade e estímulo.

À minha noiva Adriana Maria Nolasco, pelas revisões de texto e pelo carinho dedicado.

Às pessoas que colaboraram tecnicamente com esta tese: João Eduardo Ferreira e Carlos Fernando Paniago.

Aos amigos do NTIA e do CTI (Centro Tecnológico para a Informática), pelo convívio, apoio e incentivo. Em especial aqueles que colaboraram diretamente com este trabalho: Paulo César, Hsieh, Kleber e Leila.

SUMÁRIO

	Página
LISTA DE FIGURAS	- v -
RESUMO	- vi -
SUMARY	- vii -
1 INTRODUÇÃO	- 1 -
2 LINGUAGENS E MODELOS DE DADOS DE INTERESSE	- 5 -
2.1 - Considerações gerais	- 5 -
2.2 - Modelo Relacional	- 5 -
2.2.1 - "Structured Query Language" - SQL	- 8 -
2.2.1.1 - Consultas	- 10 -
2.2.1.2 - Definição de dados	- 13 -
2.2.1.3 - Manipulação da base de dados	- 14 -
2.2.2 - Implementações da Linguagem SQL	- 16 -
2.2.2.1 - Visão geral da implementação	- 16 -
2.2.2.2 - Utilização de ferramentas de programação	- 17 -
2.3 - Modelo Relacional da Tasmânia - RM/T	- 18 -
2.4 - Linguagens para base de dados orientadas a objetos	- 20 -
2.4.1 - Linguagens semelhantes a SQL	- 23 -
2.4.1.1 - PostQuel	- 23 -
2.4.1.2 - "Extensible Calculus with Entity and Set Support" - EXCESS	- 26 -
2.4.2 - Linguagens baseadas em linguagens de programação	- 28 -
2.4.2.1 - Opal	- 29 -
2.4.2.2 - "Verso Object-Oriented Data Model" - VOOD	- 32 -
2.4.2.3 - Signo-C	- 33 -
2.4.3 - Linguagens declarativas	- 34 -
2.4.3.1 - "Iris Programming Language" - IPL	- 34 -
2.4.3.2 - "GENESIS Data Language" - GDL	- 36 -
2.4.4 - Linguagens baseadas em Modelo de Dados	- 37 -

2.4.4.1 - "Entity-Relationship Language" - ERLANG	- 37 -
2.5 - Conclusão	- 40 -
3 MODELO DE REPRESENTAÇÃO DE OBJETOS	- 42 -
3.1 - Considerações gerais	- 42 -
3.2 - Aspectos genéricos do modelo	- 42 -
3.3 - Elementos do modelo de interesse neste trabalho	- 43 -
3.3.1 - Objetos	- 44 -
3.3.2 - Relacionamentos	- 45 -
3.3.3 - Atributos	- 46 -
3.3.4 - Colônias	- 48 -
3.3.5 - Meta-Base de dados	- 49 -
3.3.6 - Agregação e Generalização	- 50 -
3.3.7 - Controle de acesso, proteção e versões	- 51 -
4 LINGUAGEM DE ACESSO AO MRO	- 52 -
4.1 - Considerações gerais	- 52 -
4.2 - Abordagem da linguagem	- 52 -
4.3 - Metodologia utilizada	- 54 -
4.4 - Características da linguagem de acesso ao MRO	- 56 -
4.4.1 - Considerações gerais sobre a linguagem	- 56 -
4.4.2 - Esquema de dados usado nos exemplos da linguagem	- 57 -
4.4.3 - A notação utilizada	- 57 -
4.4.4 - Símbolos especiais da Linguagem	- 59 -
4.5 - Especificação da linguagem	- 59 -
4.5.1 - Inclusão	- 60 -
4.5.1.1 - Inclusão de elementos do Modelo no Esquema de Dados . . .	- 60 -
4.5.1.2 - Inclusão de elementos do Modelo na Base de Dados	- 68 -
4.5.2 - Eliminação	- 76 -
4.5.2.1 - Eliminação de elementos do Esquema	- 76 -
4.5.2.2 - Eliminação de elementos da base de dados	- 80 -
4.5.3 - Substituição	- 84 -
4.5.3.1 - Colônia	- 85 -
4.5.3.2 - Subtipo	- 85 -

4.5.3.3 - Multiplicidade e Segurança	- 86 -
4.5.4 - Estabelecimento de ambiente	- 86 -
4.5.5 - Seleção de objetos	- 89 -
4.5.5.1 - Gerenciamento de seleção	- 90 -
4.5.5.2 - Obtenção de seleção a partir de outras já existentes	- 91 -
4.5.5.3 - Operações válidas em seleção	- 92 -
4.5.5.4 - Acesso a cada objeto de uma seleção	- 95 -
4.5.6 - Lista/Máscara para seleções	- 96 -
4.5.7 - Mostra elemento	- 99 -
4.5.7.1 - Mostra elementos do esquema	- 100 -
4.5.7.2 - Mostra elementos da Base de Dados	- 107 -
4.6 - Considerações sobre a concepção dos comandos	- 112 -
4.7 - Conclusão	- 115 -
5 ASPECTOS DA IMPLEMENTAÇÃO DA LINGUAGEM	- 116 -
5.1 - Considerações gerais	- 116 -
5.2 - Subconjunto de comandos implementado para o GEO	- 116 -
5.2.1 - Extensão dos comandos para auxílio ao usuário	- 118 -
5.3 - Aspectos da Gramática	- 120 -
5.3.1 - Analisador sintático	- 120 -
5.3.1.1 - Prioridade e associatividade dos operadores	- 124 -
5.3.2 - Analisador Léxico	- 125 -
5.3.3 - Analisador semântico	- 127 -
5.3.3.1 - Operações sobre atributos	- 129 -
5.4 - Características da gramática para utilização do YACC	- 130 -
5.5 - Considerações sobre a utilização do YACC	- 131 -
5.6 - Conclusão	- 132 -
6 DECISÕES DE PROJETO, SUGESTÕES DE FUTURAS PESQUISAS E	
CONCLUSÕES	- 133 -
6.1 - Considerações gerais	- 133 -
6.2 - Decisões de projeto	- 133 -
6.3 - Sugestões de futuras pesquisas	- 135 -
6.4 - Conclusões	- 136 -

REFERÊNCIAS BIBLIOGRÁFICAS	- 138 -
APÊNDICE 1: Comandos LAMRO que definem o esquema da empresa de engenharia	- 144 -
APÊNDICE 2: DRO do Meta-Esquema do MRO	- 146 -
APÊNDICE 3: BNF da Linguagem de Acesso ao MRO	- 147 -
APÊNDICE 4: Conjunto de palavras reservadas da gramática	- 154 -
APÊNDICE 5: Especificação da gramática para o YACC	- 155 -

LISTA DE FIGURAS

	Página
FIGURA 1: Base de dados apoiada no RM/T	- 20 -
FIGURA 2: Exemplo de um DRO	- 44 -
FIGURA 3: Exemplo de um DRO com Relacionamento Triplo	- 46 -
FIGURA 4: Exemplo de um DRI com Relacionamento Triplo	- 47 -
FIGURA 5: Exemplo de instâncias de Colônias e Objetos	- 48 -
FIGURA 6: DRO de uma empresa de engenharia	- 58 -
FIGURA 7: Organização modular do interpretador	- 121 -

UMA LINGUAGEM PARA O GERENCIADOR DE OBJETOS BASEADA NO MODELO DE REPRESENTAÇÃO DE OBJETOS

Autor: PEDRO LUIZ PIZZIGATTI CORRÊA

Orientador: PROF. DR. CAETANO TRAINA JR.

RESUMO

Este trabalho define uma linguagem para consulta e manipulação em bases de dados, baseada nos conceitos do MRO (Modelo de Representação de Objetos), denominada LAMRO (Linguagem de Acesso ao MRO). A implementação de um subconjunto de comandos da linguagem foi realizada através de um interpretador de comandos, utilizando o sistema GEO (GEreenciador de Objetos) que suporta parte dos conceitos definidos no MRO.

A linguagem é caracterizada como uma linguagem de comandos, e situa-se entre as linguagens definidas para um modelo de base de dados específico. O processo de definição da linguagem utilizou padrões e procedimentos bem determinados para garantir que todos os conceitos do MRO estivessem presentes na linguagem.

Foram criados na linguagem dois novos conceitos: Seleções de Objetos e Máscaras. A consulta aos objetos na base de dados é realizada por comandos que manipulam o conceito de Seleções de Objetos. Estes comandos permitem elaborar consultas utilizando a semântica presente na base de dados. As Seleções de Objetos obtidas, podem ser formatadas utilizando o conceito de Máscara. Os comandos que manipulam a seleção de objetos com Máscaras podem ser caracterizados como geradores de relatórios.

A QUERY LANGUAGE TO OBJECT MANAGER (GEO) BASED IN OBJECT REPRESENTATION MODEL (MRO)

Author: PEDRO LUIZ PIZZIGATTI CORRÊA

Adviser: PROF. DR. CAETANO TRAINA JR.

SUMMARY

This work defines a language for database queries, based on the MRO's (Object Representation Model) concepts, which is named LAMRO (Access Language to MRO). The implementation of a subset of commands for that language was performed through a command interpreter, using the GEO (Object Manager) system which supports partially the concepts defined in MRO.

The LAMRO is characterized as a command language, and it is placed among the languages defined for a specific data base model. Well defined procedures and standards were used in the process of defining this language, which assured that every MRO's concepts were presents in the language.

Two new concepts were created: Object Selection and Mask. The objects queries in the data base is performed by commands that manipulate the concept of Object Selection. These commands allow to elaborate queries using the semanthics present in the data base. The obtained Object Selection can be formated thorough the concept of Mask. The commands that permit Object Selection through Masks can be characterized as report generator.

Capítulo 1

INTRODUÇÃO

Desde que a gerência da informação armazenada em alguma memória auxiliar deixou de ser exercida por programas aplicativos, e passou a ser exercida por algum Sistema de Gerenciamento de Base de Dados (SGBD), verificaram-se as vantagens que esta abordagem trouxe. Na medida em que ocorreu a centralização do controle da informação, foi possível uma independência dos dados em relação aos aplicativos, e um controle maior de consistência [Date_86], beneficiando as aplicações que necessitam manipular as informações armazenadas.

Devido à necessidade de captar os dados do mundo real, foi necessário estruturar a representação dos dados, iniciando-se estudos de Modelagens de Dados e construção de SGBDs que implementassem estes conceitos. Assim surgiram os Modelos Hierárquico (um SGBD correspondente é o IMS [Date_86]), de Rede (um SGBD correspondente é o IDMS [Date_86]) e o Relacional (um SGBD correspondente é o Sistema-R) [Codd_70, Date_86]. Estes sistemas adequam-se bem às aplicações ditas convencionais, tais como: Controle de Estoque, Contas à Pagar, etc. Já outras aplicações ditas não convencionais, tais como: Sistemas Geográfico de Informação [Roberts_91], Hipertextos, Projetos Auxiliados por Computador, Automação de Escritórios entre outros [Melo_88]; necessitam de SGBDs com características especiais, os quais são denominados em [Traina_86] como Sistemas de Base de Dados para Engenharia (SGDE). Com esses novos tipos de aplicações não convencionais, novos modelos de dados foram surgindo tentando incorporar seus requisitos. Estes modelos têm por finalidade estabelecer uma maior capacidade semântica nas representações dos dados, objetos (entes, eventos, coisas, conceitos), não prevista nos modelos anteriores.

Vários sistemas foram implementados baseados em conceitos desenvolvidos nos modelos de base de dados orientados a objetos. Entre eles pode-se citar: Postgres [Stonebraker_86], Exodus [Carey_88], GenStone [Copeland_84], GEO [Traina_91], etc.

Sistemas baseados em modelos de dados orientados a objetos assumem novas características adquiridas pela maior capacidade semântica em expressar os dados. Em

[Traina_88] são descritas características que as implementações (SGDEs) adquirem quando são baseadas no Modelo de Representação de Objetos (MRO):

"Atendem a um amplo espectro de aplicações, tais como aplicações integradas de projeto e produção de sistemas em geral;

Permitem a construção de meta-sistemas onde a especificação e requisitos da própria aplicação devam ser mantidos na Base de Dados;

Aproveitam-se das vantagens advindas do aumento da eficiência de arquiteturas de processamento paralelo".

O MRO é um modelo de base de dados orientado a objetos, definido em trabalho de doutorado desenvolvido por [Traina_88], no Instituto de Física e Química de São Carlos da Universidade de São Paulo. A formalização deste modelo foi feita a partir do acúmulo de experiências adquiridas no desenvolvimento dos seguintes sistemas: Sistema de Apoio por Computador à Documentação de Sistemas - SACDS [Traina_82] e do Sistema Integrado de Produção de Software (SIPS) [Akharas_89]. Nestes sistemas estão implementadas bases de dados que permitem suportar a especificação de metodologias, o que caracteriza estes sistemas como Meta-sistemas.

O GEO (Gerenciador de Objetos) é um Sistema de Base de Dados Orientado a Objetos baseado no MRO. Este trabalho vem sendo desenvolvido pelo grupo de Base de Dados e Engenharia de Software do Instituto de Ciências Matemáticas de São Carlos - Universidade de São Paulo. O protótipo atual deste sistema implementa os conceitos básicos do MRO, sendo esta versão utilizada de várias formas: com objetivo didático nas disciplinas de Banco de Dados do referido Instituto, para suportar ambientes de desenvolvimento de Software [Rodriguez_90, Valêncio_90], e em outros trabalhos de pesquisas na área de base de dados entre elas pode-se citar [Ferreira_91, Calônego_91, Takai_91].

De uma forma geral a utilização de SGBDs é feita através de linguagens de Base de Dados, que estabelecem uma interface de "alto nível" entre estes sistemas e as aplicações. Nestas linguagens estão expressos todos os conceitos previstos pelo Modelo de Dados suportados pelo SGBD.

As pesquisas em linguagens de base de dados acompanharam a definição e evolução dos modelos e SGBDs. Várias linguagens foram desenvolvidas, entre elas pode-se citar: a linguagem SQL [Chamberlin_74] implementada no Sistema-R e em vários SGBDs baseados no Modelo Relacional, a linguagem PostQuel do sistema Postgres [Stonebraker_86], a linguagem

Opal do sistema GenStone [Copeland_84], a linguagem IPL do sistema IRIS [Annevelink_91], etc.

No GEO a manipulação de uma base de dados deve ser feita através da utilização de rotinas da sua biblioteca [Traina_91], ativadas em programas construídos em linguagem "C" [Kernighan_84]. Desta forma, até para pequenas manipulações na base de dados, necessita-se da utilização de várias ferramentas de programação (compilador, "linkeditor", editor de textos, depurador, etc.), restringido o acesso à base de dados somente a usuários especializados.

Outra restrição a utilização desta base de dados, vem da necessidade do usuário conhecer a grande quantidade de conceitos definidos no MRO.

A finalidade deste trabalho é preencher as lacunas, descritas anteriormente para o GEO. Para isso foi desenvolvida uma linguagem baseada no MRO, de acordo com os seguintes requisitos:

- Permitir representar os conceitos do MRO em construções sintáticas, mapeadas para as suas respectivas ações semânticas;
- Auxiliar o aprendizado dos conceitos do Modelo;
- Ser de fácil utilização;
- Permitir sua implementação seguindo o mesmo ambiente operacional do GEO.

Desta forma, no capítulo 2, são apresentados os estudos das linguagens e modelos de dados que são de interesse para este trabalho. Iniciando-se pela abordagem relacional, são ressaltados os conceitos deste modelo e de sua principal linguagem, denominada SQL, para a qual são apresentadas algumas alternativas de implementação. Em seguida são abordados aspectos incorporados ao modelo relacional para aumentar sua capacidade de representação semântica, tal como tem proposto o Modelo Relacional da Tasmânia (RM/T). Finalmente, apresenta-se a abordagem de objetos através da análise de algumas linguagens de base de dados orientadas a objetos, com o objetivo de fornecer uma visão geral sobre o assunto.

São abordados no capítulo 3, os conceitos básicos do Modelo de Representação de Objetos, visando facilitar o entendimento da linguagem.

A Linguagem de Acesso ao MRO, proposta por este estudo, é discutida no capítulo 4, onde são apresentadas as características consideradas para essa linguagem, a metodologia que orientou o seu desenvolvimento e uma descrição de seus comandos.

A descrição da implementação de um interpretador desta linguagem para o GEO, é relatada no capítulo 5.

No capítulo 6, são apresentadas as conclusões obtidas neste trabalho, organizadas da seguinte forma: decisões de projeto que não foram descritas nos capítulos anteriores, sugestões de futuras pesquisas e as conclusões gerais.

Capítulo 2

LINGUAGENS E MODELOS DE DADOS DE INTERESSE

2.1 - Considerações gerais.

Este capítulo trata das linguagens e modelos de dados que são de interesse para o trabalho. Inicialmente é estudada a abordagem relacional, devido à sua importância dentre as linguagens e modelos, ressaltando-se conceitos do modelo, e de sua principal linguagem de consulta: a SQL. São descritas alternativas de implementação desta linguagem, e aspectos incorporados ao modelo relacional para aumentar sua capacidade de representação semântica, propostos pelo Modelo Relacional da Tasmânia (RM/T). Finalmente é estudada a abordagem de objetos, através do estudo de algumas linguagens de base de dados orientada a objetos, com o objetivo de fornecer uma visão geral sobre o assunto.

2.2 - Modelo Relacional.

O Modelo Relacional [Codd_70] foi utilizado como base na implementação de muitos SGBDs, entre os quais Sistema R e Database 2 implementados pela IBM, Oracle da "Oracle Corporation", entre outros. Sua popularização ocorreu de fato quando foram desenvolvidos sistemas para microcomputadores tais como dBase da Ashton-tate, Visicalc da Software Arts. Inc., etc. A implementação destes sistemas em microcomputadores baseiam-se no modelo relacional, embora segundo [Korth_89] de forma muito primitiva para ser chamados de relacionais, sendo mais apropriadamente chamados de tabulares.

A seguir, é apresentada uma revisão geral dos elementos que constituem o modelo

relacional, os quais serão utilizados na descrição da linguagem SQL no item 2.2.1.

Relação pode ser definida informalmente como sendo uma tabela de valores, organizada em linhas e colunas, onde cada coluna é um atributo e os valores de uma coluna estão definidos em um domínio de valores válidos [Date_86]. Mais precisamente o conceito de relação está fundamentado na Teoria de Conjuntos [Codd_70], da seguinte maneira:

Definição: Seja R uma relação; seja $D_1, D_2, \dots, D_n$ domínio de conjuntos e $d_1, d_2, \dots, d_n$ valores de tais domínios respectivamente. R é o conjunto de **n-tuplas** $\langle d_1, d_2, \dots, d_n \rangle$. Onde o grau da relação é n .

Exemplo: considere a tabela de Funcionários:

Número	Nome	Cargo	Endereço
100	Paulo	Professor	R. 1
200	João	Secretário	Av. 2
300	Marcos	Operador	R. 4

Através deste exemplo, serão enunciados os principais conceitos que constituem o modelo relacional, destacados a seguir:

Os nomes das colunas (Código, Nome, Cargo e Endereço) são chamados de **Atributos** da relação, para cada um dos quais está definido um **Domínio**. Exemplo: o atributo número pode ter valores inteiros situados no intervalo aberto entre 0 e 1000.

Cada linha da tabela representa uma **Tupla** e a quantidade de tuplas numa relação é a sua **Cardinalidade**. No exemplo a cardinalidade da relação "Funcionário" é 3.

O Esquema de uma relação é representado pelo seu nome, seguido pelos nomes dos Atributos entre parênteses:

Funcionário(Número, Nome, Cargo, Endereço).

As **Formas Normais** [Meier_83] tratam das dependências entre os Atributos, sendo definidos vários tipos de normalizações. Uma normalização muito importante para a construção de relações definidas para um SGBD é a chamada 1ª forma normal. As relações podem ou não estarem na 1ª forma normal. Uma relação está na 1ª forma normal quando para cada valor de atributo de cada uma das tuplas pertencentes a relação, existir somente um valor

e não um conjunto de valores (Multivalorado), e seus atributos não deverão ser compostos (subdividido em outros). Quando estas condições forem satisfeitas para todas as relações de uma Base de Dados, pode-se classificar como Base de Dados Relacional em 1ª Forma Normal.

O **Ítem de Busca (Chave)** é um conjunto de atributos, presentes em uma relação, através dos quais pode-se identificar as tuplas de uma relação. Na relação "Funcionário" o atributo "Número" tem essa propriedade, o qual será representado na descrição do esquema da relação sublinhado; então "número" é tido como **Chave Primária** da relação. O atributo "nome", também poderia ser usado para distinguir tuplas, neste caso este atributo pode ser considerado como **Chave Alternativa**.

Quando um dos atributos de uma relação é chave de outra relação, este atributo é chamado de **Chave Estrangeira**. Desta forma torna-se possível representar associações entre relações.

Existem duas Regras de Integridade para as chaves:

Regra de Integridade 1, garante que:

- a) Nenhum dos valores componentes da chave primária pode ser nulo;
- b) Uma chave identifica univocamente uma tupla na relação (unicidade).

Regra de Integridade 2, garante que a chave estrangeira pode ter valor nulo, caso contrário, este valor de chave estrangeira deve identificar uma tupla em outra relação.

Os SGBDs Relacionais devem implementar os conceitos descritos, suportando a especificação de conjunto de dados e as Restrições de Integridade, através de uma **Linguagem de Definição de Dados (DDL)** e operações através de uma **Linguagem de Manipulação de Dados (DML)**.

Por exemplo, a definição do esquema relacional da relação "Funcionário", através da DDL do Sistema-R [Date_86] pode ser:

```
DOMAIN Número    CHARACTER(3) PRIMARY
DOMAIN Nome       CHARACTER(30)
DOMAIN Endereço   CHARACTER(20)
```

```
RELATION Funcionário( FC#:  DOMAIN Número,
```

FNome: DOMAIN Nome,
FEndereço: DOMAIN Endereço)
PRIMARY KEY (FC#)
ALTERNATIVE (FNome)

Apesar de serem considerados relacionais, determinados SGBDs, entre eles o Sistema-R, não suportam alguns de seus conceitos [Date_86], tais como:

Domínios: estes Sistemas não dispõem de mecanismos para definição de tipos, intervalos, etc., os quais possibilitam definição de conjuntos de dados, a não ser os pré-definidos, tais como caracter ou numérico;

Regra de Integridade 1 não é obrigatória: atributos chaves podem ou não admitir valores nulos, no caso de admitirem em um determinado momento, só haverá uma ocorrência deste valor no atributo devido ao fato de que para cada valor de atributo chave está definido um único índice;

Regra de integridade 2 não é suportada: não se proíbe referência a tuplas inexistentes de outras relações. Este gerenciamento não é implementado devido a um aumento de complexidade do Sistema, e a perda de eficiência da atualização.

2.2.1 - "Structured Query Language" - SQL.

Desde que o Modelo Relacional foi proposto por Codd no início da década de 70, várias linguagens foram definidas para este modelo, dirigidas para usuários não experientes. Uma delas foi a Linguagem SEQUEL (Structured English Query Language) [Chamberlin_74, Meier_83], também conhecida como SQL, baseada em uma linguagem anteriormente denominada SQUARE [Boyce_73, Meier_83] sendo as duas essencialmente a mesma, embora a SQUARE utilize uma linguagem bem mais matemática, enquanto a SQL é mais parecido com o idioma Inglês [Date_86]. O Sistema-R [Date_86] utiliza a Linguagem SQL [Chamberlin_76]. Outras linguagens podem ser citadas tais como: QUEL (Query Language) [Stonebraker_76],

implementada pelo SGBD INGRES e a QBE (Query By Example) [Date_75, Meier_83] na qual o usuário manipula as relações diretamente, preenchendo gabaritos que representam as tuplas.

A descrição sobre a Linguagem SQL, apresentada a seguir, é baseada em [Chamberlin_76], da forma como é utilizada pelo Sistema-R.

A Linguagem SQL permite ao usuário:

- a) Operações de manipulação de dados tais como inserção, remoção e modificação de tuplas ou conjunto de tuplas;
- b) Definição de relações e várias alternativas de visões;
- c) Fazer controle dos dados, através da definição de regras que permitem manter a integridade dos dados;
- d) Definir transações disparadas por vários eventos;
- e) Embutir-la em outra linguagem de programação, tais como: COBOL e PL/I.

A Linguagem SQL opera sobre relações que estejam na primeira forma normal [Meier_83], ou seja: os atributos não podem ser multivalorados nem compostos (subdivididos em outros atributos).

Uma descrição resumida da linguagem e de suas características será feita a seguir, através de exemplos envolvendo:

- Consultas;
- Definição de Dados;
- Manipulação da Base de Dados.

Suponha a seguinte situação, através da qual serão baseados os exemplos de aplicação da linguagem: A relação "EMP", descreve um conjunto de empregados com o seu número, nome, número do departamento, cargo, número do empregado chefe, o seu salário e a sua comissão. A relação "DEPT", descreve o conjunto de departamentos com o seu número e nome. A relação "USO" descreve as peças usadas pelos vários departamentos. A relação "FORNECIMENTO" descreve a companhia fornecedora das várias peças. As relações são as seguintes:

EMP(NUEMP, NOME, NUD, CARGO, CHEFE, SAL, COM).

DEPT(NUD, NOME).

USO(NUD, PEÇA).

FORNECIMENTO(FORNECEDOR, PEÇA).

2.2.1.1 - Consultas.

De maneira geral, uma consulta em SQL é uma instrução que segue o seguinte formato geral:

```
SELECT <atributos>  
FROM <relações>  
WHERE <predicado>
```

O objetivo de cada uma destas cláusulas é o seguinte:

SELECT listar os atributos de relações que irão compor a relação resultante;

FROM especificar de quais relações serão obtidos os valores dos atributos especificados na cláusula anterior;

WHERE estabelecer uma coleção de predicados que compara atributos das tuplas com valores (ex. SAL < 1000) ou com outros atributos de tuplas (ex. SAL > COM). Para os predicados estão disponíveis os operadores lógicos AND, OR e NOT, sendo que os parenteses estabelecem a precedência.

A seguir serão listados exemplos de consultas em SQL.

C1. Encontre os nomes dos empregados do departamento 50.

```
SELECT NAME  
FROM EMP  
WHERE NUD = 50
```

O resultado desta consulta será uma seleção de nomes, os quais poderão ser repetidos. Isto pode ser resolvido substituindo **SELECT** por **SELECT UNIQUE**, pois assim serão eliminadas as tuplas repetidas da tabela resultante.

C2. No predicado da cláusula **WHERE**, pode-se utilizar um operador de pertinência a conjuntos (**IN**).

Exemplo: listar os nomes dos empregados e salários dos departamentos 25, 47 e

```
SELECT NOME, SAL
FROM EMP
WHERE NUD IN (25,47,53)
```

Nesse exemplo, para cada tupla, faz-se a verificação se o valor do atributo NUD pertence à relação de valores de mesmo domínio, sendo que esta relação pode ser resultado de outra consulta anterior.

Após o predicado da cláusula WHERE, pode-se especificar uma ordenação segundo o valor de um atributo das tuplas resultantes, acrescentando por exemplo: ORDER BY SAL.

C3. Uma série de funções de biblioteca estão disponíveis tais como:

- AVG (média);
- SUM (soma);
- COUNT (número de tuplas);
- MAX (máximo valor);
- MIN (mínimo valor).

Exemplo: encontrar a média dos salários das secretárias.

```
SELECT AVG (SAL)
FROM EMP
WHERE CARGO = "SECRETARIA"
```

C4. Uma relação pode ser dividida em grupos, de acordo com os valores de algum atributo, sendo que os demais atributos resultantes devem estar associados a alguma função. Com isso os atributos resultantes devem apresentar a propriedade de unicidade na relação. O agrupamento é estabelecido pela cláusula GROUP.

Exemplo: listar todos os departamentos e a média salarial de cada.

```
SELECT NUD, AVG (SAL)
FROM EMP
GROUP BY NUD
```

Note-se que NUEMP, por exemplo, não poderia ser acrescentado a cláusula SELECT, pois os valores de atributos, dentro de um mesmo agrupamento devem ser únicos.

C5. A relação que é dividida em grupos pode ter predicados que permitem escolher

somente certos grupos e desqualificar outros.

Exemplo: listar os departamentos que tenham a média salarial menor que 10000.

```
SELECT NUD
FROM EMP
GROUP BY NUD
HAVING AVG(SAL) < 10000
```

C6. Uma função especial (SET) é definida pela linguagem para executar a comparação entre um conjunto de valores de um atributo presente em um grupo, com outro conjunto de valores de mesmo domínio. Os operadores de comparação de conjuntos são: =, !=, [IS] [NOT] IN, CONTAINS e DOES NOT CONTAIN.

Exemplo: listar os departamentos que tenham empregados em todos os cargos possíveis.

```
SELECT NUD
FROM EMP
GROUP BY NUD
HAVING SET(CARGO) =
        SELECT CARGO
        FROM EMP
```

C7. Estão presentes na linguagem os operadores tradicionais de conjuntos [Meier_83], que também pertencem à Álgebra Relacional, tais como: INTERSECT, UNION, MINUS, etc. As tuplas repetidas da relação resultante utilizando estes operadores, são eliminadas antes da operação.

Exemplo: listar os departamentos que não tenham empregados.

```
SELECT NUD
FROM DEPT
MINUS
SELECT NUD
FROM EMP
```

Considerações gerais:

1. A fim de distinguir dois atributos de mesmo nome, mas de relações diferentes em uma consulta, basta utilizar o nome da relação antes do atributo (exemplo:

EMP.NUD). No caso em que a relação é a mesma, define-se rótulos diferentes para cada relação. Por exemplo:

```
SELECT X.NOME, Y.NOME
FROM EMP X, EMP Y
```

2. Uma relação pode ter as tuplas ou os atributos que a definem explicitados na consulta. Por exemplo:

```
SELECT NOME
FROM EMP
WHERE <CARGO, SALARIO> = (<'PROGRAMADOR', 50>)
```

3. Valores de atributos nulos ou desconhecidos podem ser referenciados através da palavra chave NULL. A tabela verdade, incluindo valores desconhecidos, usada para calcular valores lógicos do predicado é a seguinte:

AND		T	F	?	OR		T	F	?	NOT		
-----	+	-----			-----	+	-----			-----	+	
T		T	F	?	T		T	T	T	T		F
F		F	F	F	F		T	F	?	F		T
?		?	F	?	?		T	?	?	?		?

2.2.1.2 - Definição de dados.

Usuários podem criar ou destruir relações, definir visões alternativas de relações e especificar o modo de acesso físico (índice, etc.), que serão mantidos na Base de Dados. Cada SGBD tem suas estruturas de dados que implementam as instruções anteriores. Os exemplos a seguir estão voltados para características do Sistema-R, sendo que podem também ser aplicados a outros sistemas.

Exemplo: A relação "DEPT" pode ser definida da seguinte forma:

```
CREATE TABLE DEPT
    (NUD (CHAR(2), NONULL),
     NOME (CHAR(12) VAR))
```

O acesso físico a uma relação no Sistema-R, está ligada à organização dos índices de um ou mais atributos, mantidos na forma de árvore-B [Date_86], e à propriedade ("clustering")

das tuplas estarem armazenadas fisicamente próximas uma das outras.

Visões permitem que os usuários possam "visualizar" a base de dados de acordo com o seu interesse. Alterações podem ser feitas à medida que exista uma correspondência direta entre a tupla da visão e a relação armazenada.

Exemplo: A definição da visão de empregados do departamento 50, pode ser feita da seguinte maneira:

```
DEFINE VIEW D50 AS
    SELECT NUEMP, NOME, CARGO
    FROM EMP
    WHERE NUD = 50
```

Tabelas, visões ou qualquer outro objeto pode ser eliminado da Base de Dados.

Exemplo: Eliminar a visão D50:

```
DROP VIEW D50
```

2.2.1.3 - Manipulação da base de dados.

Alterações na base de dados são possíveis através de inserções, eliminações, modificações e associações:

2.2.1.3.1 - Inserção.

Uma única tupla ou um conjunto delas podem ser inseridas em uma relação, por exemplo:

```
INSERT INTO EMP (NUEMP, NOME, NUD)
    <535, 'JOAO', 51>
```

Os atributos não especificados na relação serão preenchidos com valores nulos.

A combinação com a cláusula SELECT, permite que tuplas da base de dados possam ser inseridas em uma relação já definida. Exemplo:

```
INSERT INTO CANDIDATOS:
    SELECT NUEMP, NOME, NUD, SAL
    FROM EMP
```



```
WHERE COM > 0.5*SAL
```

Nesse exemplo considerou-se que a relação "candidatos" já houvesse sido definida.

2.2.1.3.2 - Eliminação.

Através do comando de eliminação, tuplas podem ser removidas de uma relação, por exemplo:

```
DELETE EMP  
WHERE NUEMP = 561
```

2.2.1.3.3 - Modificação.

A modificação de tuplas é semelhante a eliminação, exceto que a cláusula SET é adicionada antes de WHERE para especificar o atributo que será alterado, exemplo:

```
UPDATE EMP  
SET SAL = SAL*1.1  
WHERE CARGO = 'PROGRAMADOR'
```

2.2.1.3.4 - Associação.

A criação de uma nova relação, derivada de relações já existentes, é permitida através de associações. Por exemplo: a relação de chefes será:

```
ASSIGN TO CHEFES (NUEMP, NOME, NUD, SAL)  
SELECT NUEMP, NOME, NUD, SAL  
FROM EMP  
WHERE NUEMP IN  
      SELECT CHEFE  
      FROM EMP
```

A linguagem SQL possui diversas outras construções sintáticas. Porém, aquelas apresentadas são consideradas suficientes para os propósitos deste trabalho, onde objetivou-se mostrar os recursos e conceitos do modelo relacional e a capacidade e forma de utilização na

linguagem.

2.2.2 - Implementações da Linguagem SQL.

Neste ítem são abordados dois estudos que tratam da implementação da linguagem de consulta SQL. O primeiro estudo apresenta o problema da implementação como um todo. O segundo propõe a adoção de ferramentas de programação para gerar um interpretador para a linguagem.

2.2.2.1 - Visão geral da implementação.

Foi implementado um interpretador da linguagem SQL [Astrahan_75] para um sistema protótipo XRM ("Relational Memory System"). Sua finalidade é fazer a tradução de um conjunto de instruções não procedurais (SQL) em uma sequência de comandos (procedurais) do sistema XRM.

Este interpretador é dividido em três partes principais:

"Parser": que faz a análise sintática e constrói a árvore de parser correspondente a cada instrução SQL. Cada cláusula WHERE é representada como uma árvore binária com predicados conectados por nós AND e OR;

Otimizador: nesta implementação o otimizador tem por objetivo minimizar as operações de recuperação de dados, armazenados em relações, determinando assim uma lista de tuplas que devem ser recuperadas através da análise dos predicados da árvore binária;

"Scanner": sua função é fazer a recuperação das tuplas geradas pelo otimizador, analisando os predicados. Assim pode-se gerar a saída (relação resultante) selecionando os atributos necessários das tuplas escolhidas ou então contabilizar os valores gerados pelas funções (por exemplo: SUM).

Tanto o otimizador como o "scanner" utilizam a árvore binária de predicados da cláusula WHERE gerada pelo "parser".

O interpretador mantém um catálogo de relações para a tradução entre os nomes utilizados pelos usuários para as relações, colunas e correspondentes, codificações internas do

XRM. Tipos de dados (inteiro, caracter) e outras informações sobre o domínio da coluna da relação também são armazenadas. O catálogo de relações é alterado de acordo com os parâmetros das instruções SQL que criam e destroem relações. Uma relação deve ser definida completamente antes que alguma tupla possa ser inserida.

2.2.2.2 - Utilização de ferramentas de programação.

A linguagem de consulta SQL permite ao usuário interagir com um SGBD relacional, o qual executa efetivamente as instruções. Em [Heerjee_88] trata-se do problema usando o ambiente de programação oferecido pelo sistema operacional UNIX [Coffin_88], utilizando suas ferramentas, tais como, o YACC ("Yet Another Compiler-Compiler") - gerador de analisador sintático - e o LEX - gerador de analisador léxico - [Kernighan_84] com o intuito de aumentar a produtividade e gerar rapidamente um protótipo.

As instruções da linguagem SQL escolhidas para serem implementadas são aquelas que somente executam consultas (leituras) nas relações. Partindo-se da definição da linguagem em BNF ("Backus-Naur Form"), converte-se essa definição para a sintaxe reconhecida pelo YACC (semelhante a BNF), acrescida das ações semânticas, que são expressas utilizando rotinas do SGBD e instruções da linguagem "C".

O analisador léxico utilizado pelo YACC no reconhecimento dos "Tokens" (ítems léxicos), que fazem parte da linguagem SQL, foi construído utilizando o LEX.

A implementação adotada não trata de nenhum tipo de otimização. Por outro lado, faz tratamento de erros utilizando as facilidades oferecidas pelo YACC.

Nessa particular implementação, a possibilidade de imbutir a linguagem SQL em outra linguagem hospedeira não é tratada, de modo que as instruções são interpretadas e, possibilitando assim, que as consultas sejam feitas interativamente, partindo-se de um SGBD já construído.

As vantagens da utilização do YACC apresentadas pelo autor são:

1. Em relação ao código na linguagem "C" gerado, este é eficiente e correto, embora as ações semânticas sejam de responsabilidade do usuário;
2. Facilidade de sua construção e alteração da definição da linguagem.

Em relação ao LEX, considera-o útil na construção de um analisador léxico que

possa ter um código fonte muito grande, pois facilita alterações e o código gerado não é muito maior que seria um programa equivalente construído sem o auxílio da ferramenta. Porém para pequenos analisadores sua utilização não é recomendada, pois o código gerado seria muito grande e ineficiente.

2.3 - Modelo Relacional da Tasmânia - RM/T.

O RM/T [Codd_79] é uma extensão do Modelo Relacional, desenvolvido com o objetivo de incorporar-lhe maior semântica. O RM/T introduz no modelo relacional essencialmente os conceitos descritos a seguir, segundo [Earl_85]:

"Uma base de dados é um conjunto de entidades, que estão classificadas segundo sua função, e para cada uma delas é associado um "surrogate" (identificador único). Este identificador existe independente ao nome que a entidade possa ter. No modelo relacional, as entidades são identificadas pelos valores dos campos chaves, controlados pelo usuário. Desse modo, se o nome de uma entidade muda, muda-se a própria entidade, tal como ela é reconhecida pelo SGBD. No RM/T, o surrogate é um código interno, não administrado pelo usuário, tornando-se assim um identificador permanente da entidade".

Para expressar maior semântica nas relações definidas pelo usuário, o RM/T mantém um conjunto de relações para descrevê-las, chamado de catálogo. Exemplo de alguma destas relações:

catr(*Rc, RELNAME, RELATYPE*).

Descreve as relações com seus nomes e tipos de entidades representadas por elas.

cata(*Ac, ATTNAME, USERKEY*).

Relação de atributos com seu nome e se participa das chaves definidas pelo usuário.

catd(*Dc, DMNAME, ORDERING*).

Relação de domínios com seu nome, para o qual está associado o tipo sintático dos valores, e se há algum tipo de ordenação entre seus valores.

catra(*RAc, Rc, Ac*).

Interrelaciona relações com os seus atributos.

Observação: o símbolo *c* denota que o atributo é um "surrogate".

Um conceito introduzido pelo RM/T, para tornar o modelo mais poderoso, é uma extensão dos operadores da álgebra relacional, abrangendo valores desconhecidos para o domínio de valores dos atributos das relações.

Com a finalidade de manter a integridade do RM/T, são adicionadas várias regras de integridade, além das duas definidas para o modelo relacional, como por exemplo as regras de integridade para classes de entidades.

O RM/T faz uma classificação das entidades presentes na base de dados segundo sua função:

1. **Característica:** entidade que qualifica ou descreve uma outra entidade, sendo sua existência dependente da entidade superior.

Em outras palavras, uma entidade tipo Y pode ser considerada um tipo de entidade característica da entidade do tipo X somente se:

- A correspondência de entidades de X para Y é de um para muitos e Y para X é de um para um;
- Cada instância de Y tem sua existência dependente da existência da correspondente instância de X.

Regra de integridade: Uma entidade característica não pode existir numa base de dados sem a entidade a qual descreve estar presente na base de dados.

2. **Associação:** é um relacionamento envolvendo duas ou mais entidades independentes, onde independente significa que nenhuma entidade tem sua existência dependente de qualquer outra.

Regra de integridade: somente poderá haver associações entre entidades existentes na base de dados.

3. **Kernel:** entidade que não é característica ou associação.

Exemplo: Considere que cada empregado (entidade do tipo Kernel), cujas propriedades são identificação e nome, tenha um histórico de trabalho (tipo de entidade característica subordinada a empregado) que tem propriedades imediatas como nome do cargo e data de início no cargo. Considere a entidade alocação (entidade do tipo associação) que associa empregados a projetos (tipo kernel). Uma base de dados apoiada no RM/T para essa aplicação, contém as relações descritas na figura 1, onde as relações "Empregado" e "Projeto" são relações de entidades Kernel, "Nome_id_empregado", "Histórico_trabalho", "Projeto_nome"

e Empregado_projeto são relações de entidades Características e "Alocação" é uma relação de entidades Associação.

Empregado		Nome_id_empregado		
Empç		Empç	ID	Nome
@		@	77	Paulo

Histórico_trabalho			
Empç	Cargo	Data	
@	c1	1/1/90	
@	c2	1/1/91	

Projeto		Projeto_nome	
Proç		Proç	Nome
\$		\$	P1

Alocação		Empregado_projeto		
Aloç		Aloç	Proç	Empç
&		&	\$	@

Figura-1: Base de dados apoiada no RM/T.

2.4 - Linguagens para base de dados orientadas a objetos.

As pesquisas em linguagens para base de dados vêm acompanhando a definição e a evolução dos modelos. Assim as primeiras linguagens foram definidas principalmente no final da década de 60 e durante a década de 70. Neste período também foram definidos os modelos tradicionais de base de dados. Dessa fase podem-se citar alguns exemplos de linguagens [Date_86], [Korth_89]:

- No modelo relacional, a principal representante é a linguagem SQL (estudada no item anterior) implementada em vários SGBDs, entre eles o Sistema R e Database 2 da IBM e o Oracle da Oracle Corporation; a linguagem QBE ("Query By Example") implementada em um sistema da IBM que leva o mesmo nome e a linguagem QUEL, desenvolvida juntamente com o SGBD Ingres [Stonebracker_76] na Universidade de Berkeley;
- Para o modelo de redes foi definido um padrão de linguagem de manipulação de dados através do grupo DBTG ("Data Base Task Group") da CODASYL. Algumas implementações seguiram a sintaxe proposta como é o caso da linguagem do IDMS desenvolvido pela Cullinane Database Systems, outras não, como no caso do SGBD Total em que a sintaxe é diferente, mas as funções executadas pelas instruções são semelhantes;
- Para o modelo hierárquico destacam-se a linguagem DL/1 do SGBD IMS (Information Management System) desenvolvido pela IBM e a linguagem do SGBD System 2000, desenvolvido pela MRI Corporation.

Uma característica comum entre as linguagens citadas anteriormente é que, normalmente, são utilizadas embutidas em linguagens de programação hospedeiras, tais como, Cobol, PL/1 ou Fortran, já linguagem QUEL é embutida em programas escritos em linguagem "C".

Na década de 80 as pesquisas em linguagens de base de dados tenderam a absorver os novos conceitos oriundos dos modelos orientados a objetos. Assim linguagens relacionais já existentes foram extendidas com o objetivo de manipular novos conceitos, principalmente no que se refere à herança e a manipulação de relações não normalizadas. Este é um caso típico da linguagem PostQuel [Stonebraker_86], que é uma extensão de Quel. Outra linha de pesquisa foi no sentido de definir linguagens a partir dos novos requisitos exigidos pelos modelos que implementam. Nesta direção foram utilizadas linguagens de programação, tais como, C++, Smalltalk, extendendo sua sintaxe ou mesmo introduzindo novos conceitos (classes) em

linguagens de programação orientadas a objetos. Esta solução foi adotada na linguagem denominada Opal (extensão da linguagem Smalltalk) do sistema GemStone. Outra solução foi definir novas linguagens no sentido de melhor adequar-se ao modelo de objetos que representa. Esta solução foi a adotada neste trabalho.

Para fazer um estudo das linguagens de base de dados orientadas a objetos foram selecionadas algumas linguagens de acordo com a classificação abaixo:

Linguagens semelhantes a SQL. Neste item são estudados alguns casos de linguagens de base de dados que envolvam conceitos de orientação a objetos com semelhanças com SQL. As linguagens estudadas são:

- PostQuel;
- EXCESS.

Linguagens definidas a partir de linguagens de programação. Neste item estão classificadas linguagens de base de dados baseadas em linguagens de programação. Estas linguagens são definidas através de extensões ou incorporando novos conceitos na linguagem de programação, necessários para a manipulação de uma base de dados. As linguagens de base de dados estudadas, com esta característica, são:

- Opal;
- VOOD;
- Signo-C.

Linguagens declarativas. Neste item estão relacionadas as linguagens de base de dados denominadas de lógicas e funcionais. As linguagens estudadas são:

- IPL;
- GDL.

Linguagens definidas a partir de modelo de dados. Aqui foram situadas as linguagens de base de dados que apresentam suas construções dependentes do modelo que implementam. Nesta classificação situam-se as seguintes linguagens:

- ERLANG;
- LAMRO - assunto deste trabalho, estudada nos capítulos 3 e 5.

A classificação adotada para as linguagens complementa a encontrada em [Cattell_91], o qual classifica somente as linguagens definidas a partir de linguagens de programação e as que são extensões de linguagens relacionais como SQL e QUEL.

2.4.1 - Linguagens semelhantes a SQL.

2.4.1.1 - PostQuel.

PostQuel é a linguagem do SGBD Postgres [Rowe_87] [Jackson_91], projetado para ser sucessor do Ingres [Stonebraker_86]. O Ingres é um SGBD relacional, e a ele foram acrescentadas características de orientação a objetos (O-O), originando o SGBD Postgres. Essas características são:

- Herança;
- Tipos Abstratos de Dados (TAD);
- Dados do tipo procedimento;
- Identificador único para os registros, semelhante ao conceito de "surrogate" introduzido pelo RM/T [Codd_79].

Sua linguagem de consulta é denominada Postquel, desenvolvida a partir da linguagem de consulta Quel do SGBD Ingres, a qual foram acrescentados características que permitem manipular os conceitos descritos anteriormente.

A linguagem PostQuel, assim como Quel é semelhante a linguagem SQL. As principais diferenças encontram-se nas características acrescentadas para manipular as características de O-O, como poderá ser observado nos exemplos da linguagem a seguir.

Um esquema de uma relação em Postgres é especificada do seguinte modo:

```
CREATE Pessoa (Idnu = int4, Nome = char[30],  
              Data_de_aniversário = date)
```

A definição da entidade "Empregado" como uma "Pessoa", pode ser feito do seguinte modo:

```
CREATE Empregado (Nro_trabalho = int4, Salário = int4,  
                 Gerente = int4)  
INHERITS (Pessoa)
```

A cláusula INHERITS indica que "Empregado", além dos atributos declarados, também herdar todos os atributos de pessoa. Nesta cláusula pode-se especificar mais que uma

entidade, caracterizando desta forma a herança múltipla.

A recuperação dos nomes em todas as linhas incluídas na tabela "Pessoa", pode ser feito da seguinte forma:

```
RETRIEVE (Pessoa.Nome)
```

A recuperação do nome de todas as pessoas, incluindo os subtipos de pessoas, utilizando uma variável tupla, pode ser feito da seguinte forma:

```
RETRIEVE (P.Nome) from P in Pessoa*
```

A tabela "Pessoa*" descrita desta forma, será a relação resultado da união da relação "Pessoa" com todas as outras relações que herdaram seus atributos.

Postquel oferece operadores de conjunto. Considere a relação "Autônomo":

```
CREATE Autonomo (End_escritorio = char[30])  
    INHERITS (PESSOA)  
RETRIEVE (ea.Nome)  
    FROM ea IN (Empregado*  $\cap$  Autonomo*)
```

Serão recuperados os nomes dos empregados que também são autônomos.

Todos os Tipos em Postquel, mesmo os tipos básicos (pré- definidos), são definidos como TAD. A definição de um TAD especifica o nome do tipo, seu tamanho em bytes, procedimentos de entrada e saída e um valor "default". Por exemplo, o tipo básico "int4" é definido da seguinte forma:

```
DEFINE TYPE int4 IS (InternalLength = 4,  
    InputProc = CharToInt4, OutputProc = Int4ToChar,  
    Default = "0")
```

Procedimentos de entrada e saída são escritos na linguagem de programação "C".

Associado com cada TAD está um conjunto de operadores. No exemplo anterior, o operador de adição de dois inteiros é definido como:

```
DEFINE OPERATOR "+" (int4, int4) RETURNS int4
```

```
IS (Proc = Adicao, Precedence = 5,  
    Associativity = "left")
```

O procedimento "Adicao" é escrito em C e define a semântica da operação. O valor de precedência é utilizado pelo sistema para avaliar expressões.

Um tipo básico pode ser estendido usando a mesma sintaxe.

Postgres também oferece tipos construtores, tal como "arrays", que permitem definir tipos estruturados.

Embora Postgres ofereça mecanismos de TAD, este é limitado em comparação com os mecanismos encontrados em muitas linguagens de programação orientadas a objetos. Não há nenhum mecanismo de herança associado com TADs do Postgres.

Uma importante característica do Postgres é a possibilidade de armazenar um atributo do tipo procedimento.

Considere-se as relações "Projeto" e "Fornecedor":

```
create FORNECEDOR (Fnome = char[20], Status = int4,  
                  Cidade = char[15])  
create PROJETO (Projnu = char[10],  
               PossiveisFornecedores = postquel,  
               CidadeProj char[15])
```

Assim pode-se criar o "Proj1", que tem como possíveis fornecedores aqueles que estão localizados na cidade São Carlos. O "Proj2" têm como possíveis fornecedores aqueles que tem o atributo "Status" maior que 30:

```
APPEND Projeto (Projnu = "Proj1",  
                PossiveisFornecedores =  
                "RETRIEVE(F.all)  
                FROM F IN Fornecedor WHERE  
                F.cidade = "São Carlos"  
                CidadeProj = "São Paulo")  
APPEND Projeto (Projnu = "Proj2",  
                PossiveisFornecedores =  
                "RETRIEVE(F.all)  
                FROM F IN Fornecedor WHERE F.status > 30",  
                CidadeProj = "São Paulo")
```

Para recuperar os nomes de possíveis fornecedores, utiliza-se a referência a este atributo da seguinte forma:

```
RETRIEVE(P.Projnu, P.PossiveisFornecedores.Fnome)
FROM P IN Projeto
```

Uma outra forma de atributo pode ser um procedimento em Postquel, utilizando procedimentos parametrizados. Através deste mecanismo, a consulta é modificada de acordo com o valor de uma tupla. Suponha a seguinte situação, onde somente fornecedores localizados na mesma cidade do projeto poderão ser possíveis fornecedores. Desta forma será necessário criar o tipo "FornecedoresDaMesmaCidade":

```
DEFINE TYPE FornecedoresDaMesmaCidade IS
  RETRIEVE(NomeFornecedor = F,Fnome)
  FROM F IN Fornecedor
  WHERE F.Cidade = $.CidadeProj
```

No exemplo anterior, "\$" refere-se a tupla corrente.

Para definir-se a relação "Projeto", pode-se utilizar o novo tipo de atributo da seguinte forma:

```
CREATE Projeto (Projnu = char[10],
  PossiveisFornecedores = FornecedoresDaMesmaCidade,
  CidadeProj char[15])
```

Estes dois mecanismos mostrados anteriormente permitem ao Postquel tratar um relacionamento como se ele fosse um atributo de um objeto.

2.4.1.2 - "Extensible Calculus with Entity and Set Support" - EXCESS.

EXCESS ("EXtensible Calculus with Entity and Set Support") é a linguagem de consulta do SGBD EXODUS [Carey_88]. Este sistema representa uma extensão de vários SGBDs, entre eles o PostGres, no sentido de aperfeiçoar o modelo relacional no que refere-se a suportar objetos complexos e novos tipos de dados.

A linguagem deste sistema é composta de duas sub-linguagens. A linguagem de definição de dados (esquema) denominada EXTRA ("EXtensible Type for Relation and

Atributes") e a linguagem de consulta denominada EXCESS ("EXtensible Calculus with Entity and Set Support").

A linguagem de definição de dados, EXTRA, é semelhante a PostQuel, acrescentando as seguintes características:

- A linguagem permite a declaração de tipos de entidades separada da declaração do conjunto de entidades. Isso pode ser observado no próximo exemplo:

```
DEFINE TYPE Pessoa:
    ( Idnu: int4,
      Nome: char[],
      Data_de_nascimento: Date )
DEFINE TYPE Empregado:
    ( Nro_trabalho: int4,
      Salario: int4,
      Gerente: ref Empregado )
INHERITS Pessoa
CREATE Empregados: {own ref Empregado}
CREATE DezMaioresSalarios: array [1..10] of ref Empregado
```

- Permite utilização dos construtores básicos de tipos: "tupla", "set" e "array", mostrados no exemplo anterior através de "Pessoa", "Empregados" e "DezMaioresSalarios". Além destes outros construtores de tipos estão disponíveis: "own", "ref" e "own ref", os quais são usados para construção de objetos complexos:

.ref: declara uma referência para outro objeto. Utilizado no exemplo anterior no atributo "Gerente";

.own: quando não é seguido de "ref", indica que por exemplo um atributo tem um valor que é um objeto (sem identificador). Seguido de "ref" será uma referência para um objeto, com identificador, sendo que a sua manipulação é feita através do objeto proprietário.

- Possibilidade de definição de novos Tipos Abstratos de Dados (TAD), como em Postgres, podendo estender os construtores de tipos. A linguagem projetada especialmente para esta finalidade é a "E", semelhante a C++ [Stroustrup_86].

- Permite a definição no esquema de procedimentos e funções, o que de forma geral é uma sequência de expressões em EXCESS, sendo que a distinção entre eles é que procedimento não retorna valor. Funções são normalmente utilizadas para definir atributos que podem ser obtidos através de outros atributos. Procedimentos são utilizados para executar alguma operação comum na base de dados.

- Permite múltipla herança, como em PostQuel.

A linguagem de manipulação (inclusão, substituição e remoção) da base de dados é similar a PostQuel, tendo desta forma as construções semelhantes a SQL.

2.4.2 - Linguagens baseadas em linguagens de programação.

Nas linguagens de base de dados que são definidas a partir de linguagens de programação, devem ser considerados alguns aspectos que encontram-se presentes nestas linguagens: Computabilidade e Persistência.

A **Computabilidade** é a característica que garante a expressividade da linguagem em representar qualquer função computável. Linguagens de base de dados definidas desta forma adquirem esta característica.

A **Persistência** [Richardson_89] é a característica relacionada com o tempo de vida de um objeto. Em linguagens convencionais, tais como C, Fortran, Pascal, etc, o tempo de vida de um objeto está ligado a duração da execução de um programa. O acesso aos dados armazenados por longo prazo é fornecido por construtores extra-linguagem, utilizando chamadas do Sistema Operacional. Linguagens persistentes garantem que os objetos sobrevivam após a execução do programa. Além disso, o programador pode manipular os objetos persistentes com expressões normais da linguagem, isto é, as operações de entrada/saída ficam transparente ao programador. Desta forma, os objetos persistentes podem ser manipulados utilizando a mesma sintaxe dos objetos voláteis.

Deve-se esclarecer que o fato de uma linguagem implementar a persistência na manipulação dos objetos não garante que características fornecidas por Sistemas de Base de Dados estejam presentes [Sutherland_89]. Algumas destas características podem ser citadas a seguir:

- A capacidade de manipular grande volume de dados armazenados com alto

desempenho;

- Um Modelo de Dados poderoso;
- A possibilidade de executar consultas ("queries") na base de dados;
- A utilização simultânea por parte de várias aplicações (Multiusuário);
- O suporte a base de dados distribuídas.

Outro aspecto importante que deve ser considerado é que esta categoria de linguagem permite eliminar a "impedância" [Jackson_91] verificada na utilização de duas linguagens de abordagens diferentes em um mesmo programa, onde detecta-se uma incompatibilidade entre a linguagem de programação hospedeira e a linguagem de base de dados embutida. Assim a "impedância" é evitada, pois nesta categoria de linguagem a manipulação de objetos é feita de um modo uniforme, isto é, utilizando a mesma linguagem.

2.4.2.1 - Opal.

Opal é a linguagem do SGBDO (Sistema de Gerenciamento de Base de Dados orientado a Objetos) GemStone [Copeland_84] [Jackson_91] que é um produto comercial, baseado na especificação da linguagem de programação orientada a objetos Smaltalk-80 [Goldberg_83], estendendo suas características, de forma a atender as necessidades de um sistema de base de dados, caracterizando assim a linguagem Opal.

Utilizando o modelo de objetos já fornecido pela linguagem Smaltalk, no que refere-se a manipulação de objetos e métodos, foram acrescentadas características envolvidas com a persistência de objetos. Como a linguagem SmalTalk é computacionalmente completa não foi considerado necessário acrescentar uma linguagem de consulta, como nas abordagens de linguagens que utilizaram-se estruturas semelhantes a SQL, estudadas anteriormente.

O modelo consiste basicamente nos seguintes conceitos: Uma classe de objetos é essencialmente um Tipo Abstrato de Dados (TAD), sendo que um objeto é uma instância de um TAD. Um objeto só pode ser manipulado via uma interface pré-definida. Esta interface consiste de um conjunto de métodos. Os métodos são invocados quando uma mensagem é enviada a um objeto. Uma mensagem contém o nome do método, o objeto destino e outros valores de parâmetros do método. Uma mensagem enviada retorna um objeto após a sua execução.

Objetos complexos são construídos através de atributos. Atributos neste modelo, são denominados **instância de variável**. Cada instância de variável pode armazenar algum outro

objeto, podendo este objeto ser atômico quando este for um caracter ou um número real.

O exemplo a seguir ilustra a declaração dos tipos "Pessoa" e "Empregado":

```
Object subclass 'Pessoa'
  instVarNames: #['Idnu', 'Nome',
                  'Data_de_aniversario']
  constraints:  #[  #[#Idnu, Integer],
                  #[#Nome, String],
                  #[#Data_de_aniversario, Date]].
Set subclass: 'ConjuntoPessoa'
  constraints: Pessoa.
Pessoa subclass: 'Empregado'
  instVarNames:
    #['Nro_trabalho', 'Salario', 'Gerente']
  constraints:  #[  #[#Nro_trabalho, Integer],
                  #[#Salario, Integer],
                  #[Gerente, Empregado]].
ConjuntoPessoa subclass: 'ConjuntoEmpregado'
  constraints: Empregado.
```

Nesse exemplo as definições das classes de objetos são feitas de acordo com a hierarquia de classes, onde "Pessoa" é declarada como uma subclasse de "Objeto". As variáveis de uma instância "Pessoa" (atributos) são "Idnu", "Nome" e "Data_de_aniversário". As instâncias da variável são restringidas a pertencer a determinadas classes. "ConjuntoPessoa" é uma subclasse de conjunto ("set"), determinando desta forma uma coleção de objetos da classe "Pessoa".

"Empregado" é uma subclasse da classe "Pessoa" e herda todas as instâncias de variáveis, as quais são adicionados outros atributos - GemStone não suporta herança múltipla. A instância de variável "Gerente" é restrita a receber valores que são instâncias da subclasse "Empregado". Suponha que "Emp" é uma instância da classe "Empregado", então "Emp.Gerente.Nome" referenciará ao nome do gerente de um empregado, o que caracteriza a construção de relacionamento em GemStone.

Toda classe tem um conjunto de métodos associados a ela. Estes são essencialmente procedimentos que criam, eliminam e fazem outros tipos de manipulações nas instâncias de objetos da classe. Por exemplo, "Pessoa" terá um conjunto de métodos responsáveis por associar valores às suas instâncias. O método que associa um valor ao atributo "Idnu" de

pessoa é definido da seguinte forma:

```
method Pessoa
    armazenaIdnu: n
    Idnu := n
%
```

Desta mesma forma deverão ser definidos outros métodos para cada um dos atributos.

Assim pode-se criar uma instância de "Pessoa", da seguinte forma:

```
Emp := Empregado new.
Emp armazenaIdnu: 1; armazenaNome: "Paulo";
    armazenaData_de_nascimento: 10/01/65.
```

Neste exemplo, a primeira linha é responsável por criar uma instância de "Empregado" através do método "new", que é pré- definido e pertence a classe "Objeto". Na segunda linha é enviada uma mensagem ao objeto "Emp", a qual especifica os métodos seguidos de seus parâmetros. Estes métodos inicializarão os atributos da instância.

O próximo exemplo ilustra a definição de um método que realiza a criação e a inclusão de uma instância da classe "Pessoa" ao conjunto "ConjuntoPessoa".

```
method: ConjuntoPessoa
    inclueIdnu: id
    inclueNome: no
    inclueData_de_aniversario: da
    | PessoaNova |
    PessoaNova := Pessoa new.
    PessoaNova armazenaIdnu: id; armazenaNome: no;
        armazenaData_de_aniversario: da.
    self add: PessoaNova.
%
```

No método acima a variável temporária "PessoaNova" é uma instância da classe "Pessoa". Para incluir valores aos atributos de "Pessoa", são utilizados os métodos já definidos para a classe "Pessoa". O método "add" é pré-definido para a classe "set" e é responsável por adicionar um objeto a classe. No exemplo, o objeto a ser adicionado é o próprio objeto sendo

manipulado, especificado por "self".

Pode-se utilizar o método definido anteriormente da seguinte forma:

```
Pessoa := ConjuntoPessoa new.  
Pessoa includeIdnu: 1; includeNome: "Paulo";  
includeData_de_aniversario: 01/10/65.
```

Para recuperar objetos, existem métodos pré-definidos tanto para a linguagem Opal, como aqueles já fornecidos pela linguagem Smalltalk, para a classe "set". Também é possível adicionar novos métodos. Considere o seguinte exemplo para um dos métodos pré-definidos:

```
Maiores_de_sessenta := Populacao select  
[:c | (c leData_de_aniversario) < 01/01/31].
```

Na avaliação desta expressão, "c" será associado a referência de cada objeto do conjunto "Populacao". Se a aplicação do método "leData_de_aniversario" retornar uma data inferior a 1931, o objeto corrente "c" é adicionado ao conjunto "Maiores_de_sessenta" que é da classe "ConjuntoPessoa".

2.4.2.2 - "Verso Object-Oriented Data Model" - VOOD.

VOOD ("Verso Object-Oriented Data Model") é um modelo de dados orientado a objetos, implementado em um SGBD denominado VERSO [Barbedette_86]. A linguagem utilizada é dividida em duas: a linguagem de definição de dados, através da qual permite-se definir os objetos e a hierarquia de classes presentes em um esquema; e a linguagem de manipulação de dados, que é baseada na linguagem Smalltalk.

O modelo de dados é baseado nos seguintes conceitos:

- Tipo de objeto: é um descritor que especifica a estrutura de suas instâncias junto com as operações permitidas;
- Classe: permite associar um tratamento de conjunto a instâncias de um determinado tipo de objetos.

Outro ponto que caracteriza este sistema e, de uma forma geral, os sistemas que são baseados em linguagens orientadas a objetos, é que a implementação de todos os elementos

previstos pelo modelo é feita através de objetos. Assim, são definidos classes de objetos para: métodos, objetos, tipos, classes, etc.

Os exemplos a seguir ilustram respectivamente a definição da entidade "pessoa", e mostra a utilização da linguagem de manipulação de dados para fazer consulta supondo-se que esta base de dados já está preenchida com instâncias desta classe.

```
TYPE          Pessoa;
SUPER_TYPE    Tuple;
STRUCTURE     BEGIN
                Nome:          string;
                Idade:          integer;
                Dependentes:    list (0,10) of Pessoa;
            end;
END_TYPE;
```

Além das definições listadas acima para o tipo "pessoa", pode-se definir ainda os métodos associados ao tipo através da palavra chave "I_METHODS".

A instrução seguinte ilustra um exemplo de utilização da linguagem de manipulação de dados, a qual tem sua sintaxe semelhante a linguagem Smalltalk. A consulta obterá uma lista de dependentes de "João":

```
Joao ?Dependentes
```

2.4.2.3 - Signo-C.

Signo-C [Lana_90] é uma linguagem de programação orientada a objetos do Sistema Signo [Salviano_92], híbrida com a linguagem C, a qual suporta uma sintaxe para manipulação de mensagens. O Signo foi desenvolvido no PFS (Projeto Fábrica de Software) [Sobrinho_87], como uma proposta de ferramenta que permita a utilização da abordagem orientada a objetos nos projetos internos do PFS.

O Signo suporta a manipulação de objetos persistentes através de uma linguagem de definição de classes e a utilização de métodos via o uso de rotinas ("msg" e "msg2"), que são responsáveis por várias tarefas, entre elas a ligação dinâmica ("late binding") do método acionado.

O sistema fornece um conjunto de classes básicas pré- definidas, tais como

"Class", "Object", etc. Nestas classes básicas estão definidos vários métodos, entre eles os que são responsáveis por consultas ("query") ao "banco de objetos" pesquisando valores de atributos declarados como "indexados" na definição da classe.

A linguagem Signo-C facilita a utilização de mensagens através de uma sintaxe pré-definida, ao invés da utilização das chamadas de rotinas do Signo, além disso, outros aspectos das facilidades dessa linguagem devem ser ressaltados:

- Simplicidade de utilização, através de parâmetros "defaults";
- Melhora a "performance" do sistema, pois introduz os mecanismos de: "Cache de métodos", através do qual reduz-se o tempo de pesquisa dos métodos que utilizam "late binding". Executa algumas verificações de tipos. E possibilita a escolha do uso de "late binding" ou ligação em tempo de compilação ("early binding").

A sintaxe básica da linguagem Signo-C pode ser generalizada da seguinte forma:

```
[(<classe>) (<id_objeto>) <método> (<parâmetros>)] ;
```

Exemplos:

1) A mensagem que executa um método responsável por exibir o atributo nome de objetos da classe pessoa:

```
[ (Pessoa) (101) mostra_nome() ] ;
```

2) Mostrar todos os nomes dos objetos da classe pessoa:

```
[ (Pessoa) (ALL) mostra_nome() ] ;
```

2.4.3 - Linguagens declarativas.

2.4.3.1 - "Iris Programming Language" - IPL.

A linguagem IPL [Annevelink_91] ("Iris Programming Language") foi desenvolvida para o protótipo de Sistema de Base de Dados Orientado a Objetos (SGBDO) denominado Iris. Esta linguagem tem a finalidade de permitir ao usuário da base de dados expressar não somente as operações básicas de manipulação de base de dados, mas também

permitir definir qualquer função computável.

O protótipo deste sistema foi desenvolvido nos laboratórios de pesquisas da "Hewlett-Packard".

O modelo de dados utilizado pelo Iris, permite a definição de tipos abstratos de dados, sendo que todo o modelo é baseado em três construtores:

- Objetos: representam entidades e conceitos do domínio da aplicação;
- Tipos: são usados para categorizar objetos em um conjunto;
- Funções: definem as ações, podendo utilizar os objetos tanto como argumento, como para retorno da função. As funções só podem utilizar objetos que pertençam aos tipos de objetos estabelecidos na definição da função.

Atributos de objetos, relacionamentos entre os objetos e computação sobre os objetos, são modelados através de funções. As funções no modelo Iris são definidas sobre tipos, podendo ser multivaloradas.

A abordagem utilizada é a funcional, permitindo operações básicas para a manipulação da base de dados, complementada com outras facilidades de linguagens imperativas de programação (Ex: "C", Pascal, etc). Algumas destas facilidades são: declarações de variáveis, instruções sequenciais, condicionais, interação e recursão.

Inicialmente foi desenvolvida a linguagem de consulta OSQL, juntamente com o protótipo do SGBDO. A estrutura sintática de OSQL foi definida para corresponder à linguagem SQL. Considerando que OSQL suporta diferenças especiais de abstração, OSQL não é super-conjunto da linguagem SQL.

Os tipos são organizados em hierarquia de tipos, desde que não gere um grafo cíclico, permitindo herança múltipla. Os tipos "agregação", ou seja, que manipulam mais de um tipo de objeto, são:

- Tupla: uma tupla é representada por uma lista não homogênea de valores, cada valor podendo ser de tipo diferente. Exemplo: [<Pessoa, String, Inteiro>];
- List: é uma lista de valores homogêneos, sendo que todos os valores herdam características de um mesmo tipo. Exemplo: lista de Inteiros.
- Bag: tem o mesmo conceito de "list", sendo que seus componentes não têm uma ordem associada.

Exemplos:

Inicialmente será criado o esquema da base de dados definindo os objetos do tipo

pessoa e a função que representa seus dependentes:

```
CREATE TYPE Pessoa PROPERTIES (
    Nome CHARSTRING);
CREATE FUNCTION Dependentes (Pessoa p) ->>
    <Pessoa, Pessoa> AS STORED;
```

A seguir será definida a função que conta o número de dependentes de uma dada pessoa:

```
CREATE FUNCTION Conta_filhos (Pessoa p) ->>
    <Pessoa, Integer> AS
    DECLARE dependentes_p;
BEGIN
    dependentes_p := Dependentes(p);
    WHILE (dependentes_p != {})
        nro_filhos := nro_filhos + 1;
        dependentes_p := REST(dependentes_p);
    END;
    RETURN [<p, dependentes_p>];
END
```

Tal como foi enunciado anteriormente, pode-se utilizar a linguagem OSQL para definir as funções. O exemplo seguinte ilustra a utilização dessa característica, definindo a função que recupera os pais que tenham um número de filhos maior que um certo valor:

```
CREATE FUNCTION Selecciona_pais (INTEGER x) ->>
    <Pessoa, Integer> AS
    SELECT Nome(p) FOR EACH Pessoa(p), Conta_filhos(p,q)
        WHERE q >= x;
```

2.4.3.2 - "GENESIS Data Language" - GDL.

Esta linguagem foi definida para o modelo de base de dados orientado a objetos GENESIS [Batory_88], sendo que uma particularidade deste modelo é a utilização de regras de produção ao invés de funções, encontradas tipicamente em modelos funcionais. Nessa linguagem, a principal característica é que as computações são expressas como fluxos de "tokens" (símbolos),

diferente das linguagens anteriores que operam conjuntos e não fluxos. Apesar da introdução destes conceitos na linguagem, também são suportados comandos relacionais, apresentando algumas construções de comandos semelhantes a SQL.

O exemplo a seguir ilustra um comando de consulta a uma base de dados, onde são listados os números de todos os empregados.

```
SELECT Emp.E#
```

Onde cada item do comando designa:

Emp: produção que devolve uma sequência de objetos do tipo empregado;

E#: produção que gera a partir de um objeto empregado um valor inteiro correspondendo ao seu número;

:: operador da linguagem para composição de produções;

SELECT: é uma macro utilizada para consulta, a qual acrescenta no final expressão de consulta outras duas produções pré-definidas pelo sistema ".NORMALIZE.1NF_PRINT", as quais são responsáveis respectivamente por: dar um tratamento de relação a sequência obtida e imprimir os símbolos gerados.

2.4.4 - Linguagens baseadas em Modelo de Dados.

Estas linguagens de base de dados caracterizam pela dependência ao modelo de dados no qual estão associadas.

2.4.4.1 - "Entity-Relationship Language" - ERLANG.

Em [Malhortra_89] é descrita uma linguagem de programação para o Modelo Entidade-Relacionamento ME-R [Chen_76].

Pode-se dizer resumidamente que o ME-R procura descrever um sistema através da representação de suas entidades, dos relacionamentos existentes entre as entidades; e dos atributos que especificam as entidades e relacionamentos modelados. Além disso, as entidades e os relacionamentos que possuem os mesmos os tipos de atributos são agrupados, constituindo os conjuntos de entidades e conjunto de relacionamentos.

A linguagem estudada para este modelo é denominada ERLANG ("Entity-Relationship LANGuage"), a qual procura estender linguagens de programação procedurais como "C", COBOL, Pascal e PL/1. Assim são definidas construções sintáticas na linguagem cuja finalidade é integrar os conceitos da linguagem de programação procedural com os conceitos inerentes do ME-R.

Esta linguagem foi implementada utilizando um SGBD relacional, utilizando desta forma várias tabelas que são definidas de maneira transparente do usuário da linguagem.

A definição do esquema da aplicação é feito através de um sistema à parte, fora do ambiente da linguagem, sendo assim não foi definida uma linguagem para definição dos dados.

A linguagem é navegacional, isto é, num determinado momento existe uma entidade e/ou relacionamento corrente.

A seguir são listados alguns exemplos de utilização da linguagem. Para isso será utilizado um esquema previamente definido com as entidades "Empregado" e "Departamento".

Pode-se incluir uma nova entidade ou relacionamento, sendo para isso necessário criar inicialmente uma instância do conjunto, e em seguida atribuir valores a ela. O exemplo abaixo inclui um empregado, um departamento e um relacionamento entre eles.

```
CREATE AN Empregado
CREATE A Departamento
CREATE A Departamento.Empregados
LET Nome(Empregado) = "Paulo"
LET Nome(Departamento) = "Computação"
LET Data_ingresso(Departamento.Empregados) = 10/10/91
```

Para eliminar entidades e relacionamentos já incluídos existem duas instruções: "DESTROY" e "DELETE", cuja sintaxe é semelhante à instrução "CREATE". A diferença entre essas duas instruções de eliminação é que a primeira verifica se o elemento a ser eliminado pertence a algum relacionamento, caso isso ocorra o elemento não é eliminado. Já a segunda elimina o elemento e os relacionamentos que estejam associados a essa entidade.

Um aspecto importante nessa linguagem é permitir manipular elementos em um conjunto através da instrução "FOR". A seguir é mostrada a sintaxe geral deste comando, onde estão representadas algumas cláusulas (opções) desta instrução.

```
FOR (EACH|ALL) <entidade>
```



```
[WITH <predicado>]
(ORDERED BY (ASCENDING|DESCENDING) <atributo>
[AND|,]) *
DO
...
LOOP
```

Através desta instrução pode-se acessar cada um dos elementos pertencente a algum conjunto de entidades. A cláusula "WITH" é responsável por selecionar os elementos desejados do conjunto, é com a qual podem-se construir predicados com expressões envolvendo valores de atributos associados ao conjunto manipulado. Na cláusula "ORDERED" é especificada a ordem que os elementos serão recuperados, onde para cada atributo que determina esta ordem, a cláusula é repetida.

Quando esta construção é utilizada para manipular um conjunto de relacionamentos, é acrescentada a cláusula "IN". Por exemplo, pode-se processar todos os empregados de um departamento específico da seguinte forma:

```
FIND Departamento WITH Nome = "Computação"
FOR EACH Empregado
  IN Departamento.Empregados(Departamento, Empregado)
  ORDERED BY ASCENDING Nome DO
...
LOOP
```

A instrução "FIND", localiza um elemento do conjunto "Departamento", cujo atributo "Nome" é "Computação", tornando assim a entidade de nome "Computação" como a corrente do conjunto "Departamento".

A instrução "FOR" está associada ao conjunto de empregados, que tenham relacionamento com o departamento de computação. A instrução "FIND" não seria necessária se na especificação do relacionamento fosse acrescentado o nome do departamento na cláusula "IN", da seguinte forma:

```
Departamento.Empregados("Computação", Empregado)
```

A instrução de controle "IF", utilizada em linguagens procedurais, também está presente na linguagem. Por exemplo, a próxima instrução avalia o valor do atributo salário das

entidades do conjunto de empregados:

```
IF Salario(Gerente(Empregado)) < Salario(Empregado) ...
```

Gerente é um relacionamento entre entidades do tipo empregado, e neste exemplo será recuperado o destino da entidade "Empregado" e o valor do atributo salário desta entidade será comparado com o valor do atributo salário da entidade "Empregado".

2.5 - Conclusão.

O estudo de linguagens de consulta e modelos de base de dados tornou-se necessário como ponto de partida para a classificação e a definição da linguagem de base de dados proposta no capítulo 4.

Entre os modelos de dados convencionais destaca-se o modelo relacional. A principal linguagem relacional é a linguagem SQL. Foi nessa linguagem que se basearam outras linguagens, principalmente nos conceitos que referem-se à definição de consultas a base de dados. Entre elas pode-se destacar Postquel, EXCESS e IPL, as quais modificam SQL através de extensões que absorveram conceitos de objetos. A linguagem SQL fornece subsídios para a definição de estruturas em linguagem de base de dados que possibilitem a consulta e manipulação de objetos.

Uma linguagem de base de dados deve fornecer toda a expressividade necessária para efetuar qualquer manipulação suportada pelo Gerenciador da Base de dados em questão. Características de computabilidade tornam-se presentes na medida em que a linguagem de base de dados é integrada juntamente com alguma linguagem de programação.

A característica navegacional da linguagem ERLANG, que define num determinado momento a existência de uma entidade ou relacionamento corrente, permite que os comandos da linguagem possam manipular o elemento corrente sem a necessidade de torna-lo explícito em um comando.

Pode-se classificar as linguagens de base de dados em 4 categorias:

- Linguagens semelhantes a SQL;
- Linguagens definidas a partir de linguagens de programação;
- Linguagens declarativas;

- Linguagens declarativas;
- Linguagens definidas a partir de modelo de dados.

As linguagens de base de dados definidas a partir de linguagens de programação tais como C++ ou Smalltalk, em geral absorveram tanto o paradigma quanto o modelo de dados definido pela linguagem de programação. A linguagem definida no capítulo 4, não segue esta abordagem, principalmente pelo fato de que esta não foi elaborada como uma extensão de uma linguagem de programação, embora tal enfoque seja possível. A solução adotada levou em conta o fato de o MRO apresentar conceitos e modelagem de dados específicos, e assim foi considerado mais interessante manter-se a independência de outras linguagens, principalmente por ser esta a primeira linguagem de consulta construída para o GEO.

Capítulo 3

MODELO DE REPRESENTAÇÃO DE OBJETOS

3.1 - Considerações gerais.

Neste capítulo são abordados de maneira sucinta os conceitos do Modelo de Representação de Objetos considerados básicos para este trabalho. O objetivo é apresentar uma descrição do modelo, para facilitar a compreensão da Linguagem de Acesso ao MRO (desenvolvida no capítulo 4), na qual estão representados estes conceitos através da sintaxe e semântica da linguagem.

3.2 - Aspectos genéricos do modelo.

O Modelo de Representação de Objetos (MRO) [Traina_88], vem atender principalmente várias aplicações ditas não convencionais, que necessitam de informações armazenadas, tais como: Projeto Auxiliado por Computador (PAC), Fabricação Apoiada por Computador (FAC ou CAM), Sistemas de Automação de Escritórios, Inteligência Artificial, Sistemas de Geográfico de Informação (SGI), etc. O modelo apresenta uma alta capacidade de representação semântica e para não comprometer o desempenho de implementações do modelo, foi concebido de tal forma que pudesse tirar proveito de arquiteturas de processamento paralelo, denominadas Máquinas de Base de Dados [Traina_86].

Sua fundamentação matemática é baseada na Teoria dos Grafos, devido ao fato que objetos podem ser tomados como nós de um dígrafo acíclico, tendo assim todos a mesma estrutura.

O nome **Objeto** para o modelo, pode ser entendido como a representação de entes, coisas e conceitos pertencentes ao mundo real, tratando-os de forma individualizada, isto é, cada

objeto mantém informações próprias, que não necessariamente precisam ser tabeladas, seguindo uma estrutura uniforme.

3.3 - Elementos do modelo de interesse neste trabalho.

Os elementos e conceitos pertencentes ao modelo destacados são: relacionamentos (binários e triplos), atributos, colônias, objetos, generalização e agregação, Meta-base de dados e controle de versões.

No esquema de dados de uma aplicação, modelado segundo o MRO, estão descritos agrupamentos de objetos, relacionamentos e atributos semelhantes, sem a preocupação de representar suas instâncias. O modelo denomina cada um desses agrupamentos de **Tipo**.

O **Diagrama de Representação de Objetos (DRO)** é utilizado para esboçar algum esquema de dados. No DRO (figura 2) estão representados tipos de objetos na metade superior dos círculos (exemplo: "Equipamento" e "Projeto"), modalidade de relacionamento no arco de circunferência (exemplo: "Utiliza/Utilizado") e o fato de que cada tipo de objeto deve habitar um tipo de colônia ("Global" e "Projeto") localizado na metade inferior dos círculos.

Figura 2: Exemplo de um DRO.

O Diagrama de Representação de Instâncias (DRI) é utilizado para esboçar as instâncias presentes num determinado momento na Base de Dados. No DRI da figura 4, estão representadas instâncias de objetos dos tipos "Peça", "Fornecedor" e "Equipamento", através de círculos; instâncias dos tipos de relacionamentos binários "Fornecida" e "Usada", através de setas unindo duas instâncias de objetos; e uma instância de um tipo de relacionamento triplo "Fornece_e_Usa", através da seta que intercepta as setas de instâncias de relacionamentos binários.

O conceito de cada um dos elementos mencionados anteriormente são tratados a seguir.

3.3.1 - Objetos.

Objeto no MRO tem um **Identificador** único que o distingue de todos os demais chamado **Nome Principal**, o qual é estabelecido no momento em que é incluído na base de dados. Portanto, através de seu nome principal um objeto pode ser acessado diretamente pelo usuário. Outra forma é utilizar um nome alternativo, um **Atributo com Característica de Sinônimo**. Este atributo tem a mesma propriedade do nome principal, isto é, ser um identificador

do objeto. Desta forma pode-se referenciar diretamente um objeto, através de seu nome principal ou pelo seu sinônimo.

Todo objeto tem um designador interno, semelhante ao conceito de surrogate do RM/T, chamado no MRO de código do objeto, o qual o identifica univocamente em toda base de dados.

3.3.2 - Relacionamentos.

Um **Tipo de Relacionamento** é caracterizado pela modalidade do relacionamento juntamente com os tipos de objetos envolvidos no relacionamento. Uma mesma modalidade de relacionamento pode ser utilizada em tipos de relacionamentos diferentes.

Através do DRO da figura 2, pode-se observar uma **Modalidade de Relacionamento** simbolizada através do arco de circunferência entre dois tipos de objetos, juntamente com o sua respectivas **Modalidade de Relacionamento Oposta**. Para cada modalidade são associadas **Multiplicidades**, ou seja, a restrição da quantidade mínima e máxima de relacionamentos dessa modalidade para cada objeto envolvido no relacionamento. Ex: a multiplicidade 1-N do relacionamento "Utiliza", indica que deverá haver no mínimo 1 e no máximo N, relacionamentos de modalidade "Utiliza" para cada objeto do tipo "Projeto".

Todo relacionamento tem um **Nível de Segurança** que determina que os objetos associados por relacionamento desta modalidade não podem ser eliminados sem que o relacionamento seja eliminado antes.

Objetos podem ser identificados por um relacionamento. Nesse caso ele é dito **Objeto Identificado por Relacionamento**, e pode ter um nome ou sinônimo que o identifique também.

As modalidades de relacionamentos vistas até agora associam dois tipos de objetos, sendo denominados de **Relacionamentos Binários**. Existem casos onde necessita-se associar três tipos de objetos ou mais num único relacionamento. Para isso o modelo fornece o **Relacionamento Triplo**.

O relacionamento triplo é formado a partir de dois tipos de relacionamentos binários já definidos, envolvendo três ou mais tipos de objetos. Os relacionamentos binários recebem um nome **Correspondente** para serem referenciados em um relacionamento triplo. No exemplo a seguir (figura 3), o relacionamento triplo "Fornece_e_Usa" é composto a partir dos tipos de relacionamentos "Usada/Usa" e "Fornecida/Fornece" , que têm como correspondentes

respectivamente "Sido_fornecida" e "Para_usar_em", entre os tipos de objetos "Peça", "Fornecedor" e "Equipamento", significando que uma peça tem "Sido_fornecida" por um fornecedor "Para_usar_em" equipamento.

Figura 3: Exemplo de DRO com Relacionamento Triplo.

Os objetos envolvidos em um relacionamento triplo são classificados em três divisões. Nas divisões um e três há somente um objeto de cada tipo e na divisão dois podem haver mais de um objeto do mesmo tipo, ou de outros tipos definidos nesta divisão. Para melhor entendimento, suponha um relacionamento triplo "Fornece_e_Usa" que associa o objeto Pc1 do tipo "peça" (divisão um), que é "fornecida" pelo "fornecedor" F1 (divisão três) que é "usada" nos "equipamentos" (divisão dois) E1, E2 e E3. O DRI da figura 4, representa graficamente esta situação.

A simbologia deste diagrama é semelhante ao DRO, sendo que uma circunferência representa a instância de um objeto e o arco entre os objetos representa a instância de um relacionamento e não um par de modalidades de relacionamentos como no DRO.

3.3.3 - Atributos.

De maneira geral, atributos são utilizados para descrever objetos ou relacionamentos. Da mesma forma que objetos podem ser agrupados em tipos, pode-se ter **Tipos**

Figura 4: Exemplo de um DRI com Relacionamento Triplo.

de Atributos, pois atributos pertencentes ao mesmo tipo devem ter a mesma descrição. Portanto os atributos estão representados no DRO através de seu tipo. Sua multiplicidade, também deve estar representada para definir atributos multivalorados.

Para cada atributo deverá estar associada a sua **Característica** (significado) escolhida entre as pré-definidas pelo modelo. As características previstas pelo modelo são:

- **Sinônimos:** indicam identificadores alternativos para objetos;
- **Propriedades:** correspondem aos tipos de dados básicos, tais como inteiros, reais, etc.
- **Comentários:** são informações longas, com estruturas não tratadas;
- **Estruturas:** são agregações de outras associações (atributos e relacionamentos);
- **Regras:** indicam a ação a ser tomada quando determinada Condição ocorre;
- **Gráficos:** indicam coordenadas geométricas;
- **Tempo:** indica um valor de tempo;
- **Procedimento:** indica uma atividade a ser exercida pelo sistema de uma forma pré-determinada;

- **Imagens:** indicam que representação da informação de forma matricial ("pixel");
- **Visualização:** figura que representa o elemento.

No DRO das figuras 2 e 6 podem ser observados tipos de atributos, sendo que cada característica é representada por um símbolo específico.

3.3.4 - Colônias.

Uma **Colônia** de objetos classifica (particiona) objetos da base de dados, podendo numa mesma colônia habitarem objetos de diferentes tipos. A única colônia que é previamente definida é a Global, sendo que a partir dela pode-se criar outras colônias. Objetos que habitam uma colônia, podem **Constringir** outras colônias de objetos como mostra a figura 5.

Figura 5: Exemplo de instâncias de colônias e objetos.

Na figura 5 os objetos P1 e P2 constringem colônias do tipo "PROJETO", sendo que na colônia "PROJETO P1" habitam os objetos E1 e E2 do tipo "Equipamento" e na colônia "PROJETO P2" habitam os objetos E3 e E4, também do tipo "Equipamento". Deste modo pode-se observar que uma colônia de objetos além de proporcionar um nível específico de detalhe constringindo objetos, permite também determinar aspectos de interesse, fazendo a partição de objetos pertencentes a um mesmo tipo de colônia.

As colônias são classificadas em **Tipos de Colônias** os quais, por sua vez, classificam os tipos de objetos. Cada objeto da base de dado, deve ser de um tipo de objeto e habitar uma colônia (vide a figura 5), e cada tipo de objeto deve habitar exatamente um tipo de colônia, como pode ser observado nos DROs das figuras 2 e 3. A hierarquia de colônias na qual um objeto é localizado é denominado de **Contexto** do objeto.

Seguindo a dinâmica do MRO, num determinado momento somente estará **Disponível** uma colônia de cada tipo, a qual é chamada de **Colônia Aberta** dentre as colônias daquele tipo. As demais colônias estarão **Fechadas**. Quando inciam-se as operações na base de dados, somente a colônia Global está disponível.

Os objetos que habitam uma colônia aberta são **Objetos Disponíveis**. Os objetos disponíveis podem ser **Considerados** para abrir as colônias que constriem. Um objeto pode constriar mais de uma colônia, desde que sejam de tipos diferentes.

A localização de um objeto na Colônia Aberta (em uso), é feita através de seu tipo seguido pelo nome principal ou pelo seu sinônimo. Somente objetos que habitam colônias diferentes podem ter o mesmo nome principal.

Para referenciar diretamente os objetos habitantes de uma mesma colônia pelo seu nome, esta deve estar aberta para acesso, naquele momento. Inicialmente somente a colônia global tem seus objetos acessíveis. Para ter-se acesso a objetos de colônia não aberta, o objeto que a constrieste deve ser **Considerado**, tornando-a aberta.

Quando um objeto é considerado para abrir uma colônia, se houver uma outra colônia do mesmo tipo aberta, esta será fechada. Outras colônias que foram abertas não pertencentes ao tipo em questão, não serão afetadas.

Como foi descrito anteriormente, o acesso a um objeto em uma colônia aberta, pode ser feito através de seu nome principal ou seu sinônimo. Para os objetos pertencentes a outras colônias diferentes da corrente (aberta), deve-se especificar sua localização através do seu **Caminho de Acesso**, ou indicando o seu contexto, isto é tornado sua colônia aberta.

3.3.5 - Meta-Base de dados.

Uma base de dados completa usando o MRO é composta de uma Meta-Base de Dados e de uma Base de Dados propriamente dita. A **Meta-Base de Dados** armazena os tipos de colônias, tipos de objetos, tipos de atributos, etc, o que caracteriza o **Esquema** da aplicação.

A Base de Dados armazena as instâncias de objetos, colônias, atributos, etc.

A Meta-base de dados é armazenada como uma colônia, cujo tipo é controlado pelo sistema, e é denominada colônia Esquema. O esquema da própria colônia esquema é pré-definido e constitui o Meta-Esquema do MRO. O apêndice 2 apresenta o DRO do Meta-esquema do MRO.

O objeto que restringe a colônia Esquema é um objeto controlado pelo sistema e habita a colônia Global, sendo do tipo β ESQUEMA. Existe somente um objeto deste tipo, porém podem existir múltiplas variantes da colônia Esquema.

O acesso à Meta-base de dados é feito da mesma maneira que o acesso a base de dados, permitindo que o esquema possa ser dinamicamente atualizado.

3.3.6 - Agregação e Generalização.

Tipos de objetos ou de relacionamentos podem ser especializados em Subtipos de Objetos ou de Relacionamentos respectivamente. Subtipos de objetos ou de relacionamentos são de um tipo básico, que cumpre um predicado sobre os seus atributos e relacionamentos. Por exemplo, no DRO do Meta-esquema do MRO (apêndice 2), pode-se observar que "Atributo não identificador" é subtipo do tipo de objeto "Atributo", onde o predicado que define o subtipo é que o seu atributo "Característica" é diferente de "Identificador".

Subtipos de objetos ou de relacionamentos podem ser refinados em outros subtipos ou reagrupados com outros tipos e subtipos, formando outros subtipos. A rede de subtipos é acíclica. Todo o subtipo tem um tipo mestre, que é o tipo ou subtipo do qual o subtipo em questão é especialização. Por outro lado, todo subtipo tem um tipo básico, o qual não é subtipo de nenhum outro subtipo. O agrupamento de subtipos em outros somente pode ser feito quando o tipo básico de tudo o que for agrupado for único.

Objetos podem ser semanticamente agrupados, formando **Supertipos**. Supertipos, ao contrário de subtipos, caracterizam outros objetos, cujos atributos sumarizam coletivamente os objetos agrupados. Por exemplo poderia ser criado um supertipo "Laboratorio" na colônia "PROJETO P1", de acordo com a figura 6, agrupando os equipamentos de tal forma a determinar a média dos atributos "Valor" dos equipamentos do laboratório.

Todos os objetos agrupados num supertipo devem ser de tipos que habitam um único tipo de colônia. Assim, como pode-se observar no exemplo anterior, um supertipo só

agrupa objetos que habitam uma mesma colônia na base de dados.

3.3.7 - Controle de acesso, proteção e versões.

O mecanismo de **Proteção de Acesso** do MRO atua a nível de colônias. Para que uma colônia seja aberta ela deve estar acessível. As colônias podem estar acessíveis para leitura ou leitura e escrita [Calônego_91, Ferreira_91].

A proteção de acesso atua em três níveis: proteção por atributo de usuário, proteção por acesso multi-usuário e **Proteção por Variantes**.

Podem existir mais que uma colônia de objetos do mesmo tipo constricta pelo mesmo objeto, porém com acesso mutuamente exclusivo. Cada uma delas é uma **Variante** da mesma colônia.

Variantes que não devem mais ser alteradas são **Congeladas**, e o sistema não permitirá mais o acesso para leitura e escrita. Tais variantes são chamadas **Versão da Colônia**.

Variantes que podem ser alteradas são chamadas de **Alternativas da Colônia**. Variantes são criadas como alternativas, e quando ficam prontas podem ser congeladas, transformando-se em versões.

Variantes já existentes podem ser Instanciadas para criar outras alternativas, portanto variantes são **Instâncias de Colônias de Objetos**.

A variante de uma colônia que um usuário tem acessível num dado instante é chamada **Variante Corrente**.

Capítulo 4

LINGUAGEM DE ACESSO AO MRO

4.1 - Considerações gerais.

Neste capítulo é apresentada a Linguagem de Acesso ao MRO proposta para o GEO que será designada como LAMRO. Inicialmente são situadas as características que foram consideradas para a elaboração da linguagem, a metodologia que orientou o seu desenvolvimento e a descrição completa dos comandos que compõe a linguagem.

4.2 - Abordagem da linguagem.

A linguagem de Acesso ao MRO (LAMRO) visa estabelecer uma comunicação de "alto nível" entre o usuário e o MRO, tendo em vista permitir sua utilização de forma interativa.

Para definir esta linguagem foram consideradas algumas características que pudessem guiar a sua definição, entre as quais podem-se citar:

- Mapear os conceitos definidos no Modelo de Representação de Objetos em construções sintáticas e suas respectivas ações semânticas;
- Auxiliar no aprendizado dos conceitos do Modelo;
- Ser de fácil utilização;
- Permitir sua implementação seguindo o ambiente operacional do GEO.

Para definir uma linguagem seguindo as características acima, foram consideradas três abordagens: linguagem natural de formato livre, linguagem orientada a "menus", e linguagem baseada em comandos e argumentos.

A primeira, apesar de ser de fácil aprendizagem e oferecer grande flexibilidade,

apresenta uma implementação cara, difícil e com gramática ambígua. Em [Mamede_88] critica-se esta solução por dois motivos: "... o sucesso obtido com linguagens naturais é normalmente alcançado em situações em que o significado das palavras é restrito a um contexto bem definido. As implementações atuais apresentam poucas funções em relação ao número de opções disponíveis em uma linguagem realmente natural (caso da linguagem falada/escrita). Estas duas razões implicam em restringir a tal ponto o vocabulário e a liberdade do uso da linguagem que ela perde sua principal característica, a de ter o formato livre".

As linguagens orientadas por "menus", assim como as linguagens naturais, permitem uma aprendizagem rápida e também evitam ambiguidades. Apresentam outra vantagem que é de reduzir o "overhead" na análise das instruções fornecidas pelo usuário, pois a informação gerada pelo usuário são orientadas pelo sistema, adequando-se desta forma a usuários iniciantes. Por outro lado, uma tela de "menu" apresenta uma série de informações desnecessárias para um usuário experiente. Portanto, essas linguagens podem gerar fluxos de informação desnecessários entre o usuário e o sistema. Para contornar este problema, existem técnicas de permitir o estabelecimento de sequências completas de navegação em um único passo. Entretanto tais técnicas apresentam o inconveniente de obrigar o usuário a memorizar as sequências de que mais necessita.

Em uma linguagem baseada em comandos e argumentos, o usuário expressa cada ação a ser executada através de uma estrutura formada pelo nome de um comando, seguido ou não pelos argumentos que o qualificam. Apresenta a vantagem de permitir a utilização dos comandos da linguagem em sequência arbitrária, o que não é permitido na abordagem anterior. A principal desvantagem desta abordagem é exigir do usuário um aprendizado anterior sobre alguns aspectos da linguagem de comando tais como: a sintaxe do comando, as palavras reservadas, notações especiais, etc. Isso torna-se um inconveniente para a aprendizagem dos usuários.

Dentre as abordagens consideradas decidiu-se optar pela linguagem orientada a comandos. Considerou-se nesta decisão além dos aspectos negativos citados para as outras duas abordagens, aspectos específicos do MRO. Por exemplo: considerando que o MRO manipula muitos conceitos, sendo cada um deles com características próprias, uma linguagem natural geraria um vocabulário com muitos contextos diferentes, tornando complexa sua implementação; e por ser restrita a contextos, tornar-se-ia similar a uma linguagem orientada a comandos. Com relação à linguagem orientada a menus, geraria uma hierarquia de menus complexa. Além do

mais considerou-se que o usuário desta linguagem já tem alguma experiência no uso do computadores, bem como no uso de uma linguagem de programação, tornando-se desnecessário direcioná-lo durante a utilização da linguagem.

Para minimizar os inconvenientes das linguagens orientadas por comandos, bem como atender as características listadas anteriormente como sendo as desejadas para a linguagem de acesso ao MRO, foram estabelecidos alguns critérios que devem ser seguidos na definição da linguagem de comandos:

- Os nomes de comandos e símbolos da linguagem devem lembrar as funções ou elementos do modelo que manipulam e ser de fácil aprendizagem;
- Manter um padrão para o formato de comandos que realizam operações equivalentes;
- Procurar minimizar o número de comandos. Por exemplo manipulando em um mesmo comando operações completas e/ou parciais;
- Permitir uma flexibilidade na utilização da linguagem, podendo haver mais de uma maneira de realizar a mesma operação. Isto permite que comandos manipulem alguns conceitos em comum. A diferença entre eles situa-se nos conceitos que são específicos de cada comando;
- Os comandos devem permitir formas simplificadas ou completas;
- Aceitar comandos fornecidos na sequência definida pelo usuário;
- Indiferença entre letras maiúsculas e minúsculas e minimização de pontuações especiais.

Uma linguagem de comandos incorporando estes aspectos oferece um formato mais próximo ao da linguagem natural. No entanto, elas ainda têm o inconveniente exigir: por parte do sistema uma validação dos comandos e argumentos digitados (análise sintática); e por parte do usuário, uma memorização das regras da linguagem.

4.3 - Metodologia utilizada.

O desenvolvimento da linguagem pode ser dividido em duas etapas: Definição da Linguagem e Implementação da Linguagem.

a) Definição da Linguagem.

Nesta primeira fase procurou-se especificar uma linguagem, tendo como ênfase abranger os aspectos conceituais do Modelo, sem a preocupação de estabelecer uma gramática robusta, ou seja, completamente especificada.

Como resultado desta fase foi escrita uma versão preliminar do documento contendo a definição da linguagem [Traina_89]. O documento serviu de base para a fase seguinte.

No item 4.5, será feita uma descrição dos resultados obtidos nesta fase.

b) Implementação da Linguagem:

Utilizando a especificação inicial da linguagem e baseando-se na biblioteca de rotinas implementadas que constitui o GEO, passou-se a preencher os aspectos gramaticais necessários para a implementação que ficaram incompletos na fase anterior. Para isso foi necessário:

- Estabelecer com rigor os critérios sintáticos da linguagem inicialmente definida, o que se traduziu em: definir palavras reservadas, separadores, operadores com suas respectivas associatividades e prioridades, estabelecer a gramática da linguagem;
- Definir novos comandos, que foram identificados como sendo necessários para a utilização da linguagem de maneira interativa. De forma geral, trata-se de comandos que possibilitam ao usuário obter informações armazenadas, bem como consultar o estado em que encontra-se a base de dados num determinado instante;
- Resolver os casos omissos que dizem respeito a ações semânticas dos comandos da linguagem. Embora na especificação da linguagem elaborada na fase anterior descrevam-se as ações de cada comando, várias ações semânticas necessitaram ser melhor especificadas e detalhadas, para permitir usar as rotinas do GEO.

Na introdução desta dissertação, deixou-se claro que este trabalho não trata da implementação do GEO, ou qualquer de suas rotinas, mas utiliza-se o que já existe para suportar

o interpretador da Linguagem. Desta forma, o interpretador da linguagem está operacional, sendo que não estão presentes os comandos que o GEO ainda não suporta.

Como resultado desta fase obteve-se uma definição da linguagem para os comandos cujos conceitos encontram-se suportados pelas rotinas do GEO já implementadas - capítulo 5.

4.4 - Características da linguagem de acesso ao MRO.

Os comandos da linguagem atuam sobre os elementos do Modelo e suas instâncias, tais como: objetos, relacionamento, atributo, colônia, subtipo, supertipo, etc. Os comandos foram classificados de acordo com a finalidade para os quais foram definidos:

Inclusão - permitem incluir a definição um novo tipo, ou a instância de um elemento do Modelo;

Eliminação - permitem eliminar algum elemento incluído;

Substituição - permitem a alteração de alguns dos elementos;

Estabelecimento de Ambiente - permitem considerar colônias a fim de tornar objetos disponíveis;

Seleção de Objetos - permitem obter agrupamento de objetos, através de consultas à base de dados;

Lista Seleção com Máscara - permitem listar objetos pertencentes a uma seleção, exibindo as informações através de um filtro (máscara);

Mostra Elemento - permitem exibir informações sobre os diferentes elementos pertencentes ao modelo.

4.4.1 - Considerações gerais sobre a linguagem.

a)Dentro da classificação adotada para os comandos, dividiu-se ainda cada conjunto de comandos naqueles que manipulam o esquema e naqueles que manipulam a base de dados propriamente dita. Esta divisão tem a finalidade de auxiliar no entendimento, não significando que estes comandos devem ser utilizados agrupados desta forma. De maneira geral os comandos do esquema não afetam o estado corrente da base. Por exemplo, isto significa que

pode-se executar comandos na seguinte sequência:

- 1 - Inclusão do objeto O1 na base (tornando desta forma este o objeto O1 corrente);
- 2 - Inclusão do tipo de objeto T no esquema (O1 permanece corrente);
- 3 - Inclusão de um valor de atributo V1 (a inclusão deste valor refere-se ao objeto corrente na base: O1).

b) Vários comandos que manipulam a base afetam o seu estado. Desta forma, a sequência com que algum deles são executados é importante, como pode-se observar na sequência do exemplo anterior onde a inclusão de um valor de atributo na base (3), está relacionada com uma operação anterior que torna um objeto corrente (1).

No entanto, aconselha-se que as operações sobre o esquema sejam feitas inicialmente, pois garante-se que não haverá inclusões de instâncias na base de dados antes de sua definição.

4.4.2 - Esquema de dados usado nos exemplos da linguagem.

Para auxiliar a explicação dos comandos da linguagem, será utilizado um esquema de dados, sobre o qual serão elaborados os exemplos da linguagem. Trata-se de um esquema simples, que modela uma empresa de engenharia segundo o MRO. Esse esquema é mostrado na figura 6 (a seguir), onde representam-se **projetos**, **funcionários** que dele **participam** e **equipamentos** utilizados durante seu desenvolvimento, as **peças** que são usadas por um equipamento, as quais são fornecidas por fornecedores. No esquema estes entes foram modelados como tipos de objetos, que associam-se segundo tipos de relacionamentos binários (**Fornece**, **Usa**, **Utiliza** e **Participa**) e um tipo de relacionamento triplo, que representa o fato que uma **peça** tem sido fornecida por um **fornecedor** para **usar** em vários **equipamentos**. Os **funcionários** por sua vez, podem ser classificados em dois subtipos: **técnicos** e **administrativos**.

4.4.3 - A notação utilizada.

Na especificação sintática dos comandos, foi utilizada a seguinte notação:

Palavras reservadas aparecem em letras maiúsculas e todos os argumentos

Figura 6: DRO de uma empresa de engenharia.

fornecidos pelo usuário em letras minúsculas;

< > delimitam algum símbolo fornecido pelo usuário;

[ ] delimitam algum conjunto de símbolos opcionais;

()* delimitam um conjunto de símbolos que podem aparecer zero ou mais vezes;

()+ delimitam um conjunto de símbolos que podem aparecer uma ou mais vezes;

| indica uma escolha possível entre argumentos mutuamente exclusivos;

/* */ delimita algum comentário explicativo sobre o comando, mas não

pertencente a ele.

4.4.4 - Símbolos especiais da Linguagem.

Na definição da linguagem, foram utilizados alguns símbolos especiais. Aqueles que são utilizados neste texto estão descritos a seguir:

- / Denota a especificação de caminho através hierarquia de colônias para referenciar um objeto. Seu emprego é semelhante ao separador / utilizado pelo "shell" (interpretador de comandos) do UNIX. Ex: um objeto do tipo equipamento pode ser referenciado a partir da colônia global por: /P1/E1;
- ; Finaliza um comando;
- + - ^ ~ Operadores de conjunto para seleção;
- . Denota o objeto corrente;
- @ Referência a seleção corrente;
- \$ Operador de restrição para seleção;
- { } Utilizado em comando de seleção para iniciar e fechar respectivamente bloco de comandos da linguagem. Pode ser utilizado para agrupar atributos ou relacionamentos de uma máscara;
- " " Delimita uma sequência de caracteres ("string");
- : Indica na inclusão de uma instância de relacionamento triplo, o início da especificação da modalidade de relacionamentos com os objetos da divisão dois.

Observação: Para os comandos da linguagem, o interpretador trata indiferentemente letras maiúsculas e minúsculas. No entanto, utilizaram-se nesta descrição os comandos sempre em letras maiúsculas para denotar palavras reservadas da linguagem, com a finalidade de facilitar sua compreensão. Note-se que as palavras definidas pelo usuário, tal como nomes de tipos, identificadores de objetos, etc. devem ser usados tal como definidos, ou seja, pode-se usar "DEFINE", "Define" ou "define", já que trata-se de um comando da linguagem. Porém "funcionário", que é definido pelo usuário é diferente de "Funcionário".

4.5 - Especificação da linguagem.

4.5.1 - Inclusão.

Os comandos de inclusão, permitem que elementos do Modelo possam ser introduzidos tanto no **Esquema de Dados** como na **Base de Dados**, tais como: objetos, relacionamentos, atributos e colônias. Para que um relacionamento ou um atributo seja incluído na base de dados, deve haver um objeto corrente. Quando um objeto ou um relacionamento é criado na base de dados, imediatamente torna-se corrente. De modo geral o comando "LOCALIZA", torna corrente um objeto, relacionamento ou colônia. Os comandos que não explicitam um objeto ou relacionamento, referenciam-se implicitamente ao objeto ou relacionamento corrente.

4.5.1.1 - Inclusão de elementos do Modelo no Esquema de Dados.

O comando "DEFINE", seguido do nome do elemento, é utilizado para incluir um elemento do modelo no esquema. Este comando não afeta o estado corrente da base de dados. Sua estrutura geral é mostrada a seguir:

Sintaxe geral:

```
DEFINE <elemento> <argumentos> ;
```

As diversas formas deste comando são detalhadas a seguir.

4.5.1.1.1 - Tipo de Objeto.

Um ou mais tipos de objetos podem ser definidos no Esquema, utilizando o seguinte comando:

Sintaxe:

```
DEFINE OBJETO (<tipo_objeto>)+  
[ IDENTIFICADO POR <modalidade_rel> <t_obj_origem>  
                  <t_obj_destino> ] ;
```

Exemplo: Incluir os tipos de objetos projeto e equipamento:

```
DEFINE OBJETO Projeto, Equipamento;
```

O novo tipo de objeto definido desta forma, por "default" pertencerá a colônia global. Na definição de um tipo de colônia pode-se posteriormente alterar a colônia na qual o tipo habita. Isto poderá ser observado a seguir, onde exemplifica-se que o tipo de objeto equipamento habita a colônia projeto, e não a colônia global.

A cláusula "IDENTIFICADO POR" é utilizada para definir que os objetos daquele tipo são identificados por um relacionamento do tipo indicado.

4.5.1.1.2 - Tipo de Colônia.

O comando a seguir, permite incluir no esquema um novo tipo de colônia, ou associar os tipos de objetos que habitam um tipo de colônia já definida:

Sintaxe:

```
DEFINE COLONIA <tipo_colônia>  
    CONSTRITA POR <tipo_objeto>  
    HABITADA POR (<tipo_objeto>)* ;
```

A definição prévia dos tipos de objetos da cláusula "HABITADA POR" no comando "DEFINE COLONIA" não é necessária, podendo o tipo de objeto ser relacionado na cláusula sem que seja antes definido através de um comando "DEFINE OBJETO". Nesse caso o tipo de objeto será criado automaticamente. Já o tipo de objeto da cláusula "CONSTRITA POR" deve ser definido previamente antes do comando de definição do tipo de colônia.

Caso o tipo de colônia já tenha sido incluído no esquema, este comando pode ser utilizado para incluir outros tipos de objetos que habitam nesse tipo de colônia.

Exemplo: Criar a colônia de projetos, para a qual um certo número de equipamentos serão alocados (considerando que um equipamento seja alocado somente a um projeto específico).

```
DEFINE COLONIA PROJETO  
    CONSTRITA POR Projeto  
    HABITADA POR Equipamento;
```

4.5.1.1.3 - Tipo de Relacionamento.

Uma modalidade de relacionamento e seus tipos de relacionamento associados podem ser definidos, através do seguinte comando:

Sintaxe:

```
DEFINE RELACIONAMENTO <modalidade_relacionamento>
    [OPOSTO <modalidade_relacionamento>]
    ( ORIGEM <tipo_objeto>
      DESTINO <tipo_objeto>
        [SEGURANCA SIM|NAO] /* assume NAO */
        [MINIMO <número>] /* assume 0 */
        [MAXIMO <número>] /* assume n */
        [MINIMO OPOSTO <número>] /* assume 0 */
        [MAXIMO OPOSTO <número>] /* assume n */
    ) + ;
```

A modalidade pode já existir ou não, e neste último caso ela é criada. Desta forma, uma mesma modalidade de relacionamento pode estar composta juntamente com diferentes pares de tipos de objetos origem e destino, denominando-se esta composição de tipo de relacionamento.

Caso a modalidade já tenha sido definida num comando anterior, a modalidade do relacionamento oposto pode ser omitida, mas se especificada, deve ser a mesma utilizada anteriormente. A modalidade deve ser especificada obrigatoriamente somente na primeira vez em que essa modalidade for definida.

As cláusulas "ORIGEM/DESTINO" devem ser especificadas para cada tipo de relacionamento associado à essa modalidade de relacionamento. As cláusulas "MINIMO/MAXIMO" especificam para cada tipo de relacionamento o número mínimo e máximo de ocorrências de instâncias de relacionamentos para o tipo de relacionamento em questão. A cláusula "SEGURANCA" diz respeito ao fato do objeto destino do relacionamento dever ou não ser eliminado, se o relacionamento deixar de existir.

Exemplo: Definir o relacionamento de modalidade "Utiliza", entre os objetos do tipo "Projeto" e "Equipamento".

```
DEFINE RELACIONAMENTO Utiliza
    OPOSTO Utilizado
    ORIGEM Projeto
    DESTINO Equipamento;
```


4.5.1.1.4 - Tipo de Relacionamento Triplo.

A definição de um relacionamento triplo no Esquema pode ser feita utilizando o seguinte comando:

Sintaxe:

```

DEFINE RELACIONAMENTO TRIPLO <tipo_rel_triplo>
    [MINIMO <número>]           /* assume 0 */
    [MAXIMO <número>]           /* assume n */
    [SEGURANCA SIM|NAO]         /* assume NAO */
    MODALIDADE UM <modalidade_relacionamento>
        CORRESPONDENTE <correspondente_triplo_d3>
            [MINIMO <número>]       /* assume 1 */
            [MAXIMO <número>]       /* assume n */
    MODALIDADE DOIS <modalidade_relacionamento>
        CORRESPONDENTE <correspondente_triplo_d2>
            (
                DIVISAO UM <tipo_objeto>
                DIVISAO DOIS (<tipo_objeto>)+
                DIVISAO TRES <tipo_objeto>
            )+ ;

```

Um relacionamento triplo é composto por duas modalidades de relacionamentos pré-definidas (*modalidade_relacionamento*), que compoem relacionamentos binários entre o objeto na divisão um e os objetos na divisão dois, e outra que compoem um relacionamento entre o objeto na divisão um e o objeto na divisão três. Para cada modalidade de relacionamento envolvida deve-se especificar a modalidade correspondente para ser referenciada no relacionamento triplo.

Note-se que a multiplicidade tanto do relacionamento triplo como da modalidade de relacionamento com o tipo de objeto da divisão dois podem ser especificadas, sendo que a modalidade de relacionamento com o tipo de objeto da divisão três tem multiplicidade pré-definida igual a um. A multiplicidade da modalidade um, indica quantos objetos podem haver na divisão dois para cada relacionamento triplo, ao passo que a multiplicidade do triplo indica quantos relacionamentos triplos do mesmo tipo podem existir tendo os mesmos objetos da divisão um e divisão três.

Exemplo: Definir o relacionamento triplo "Fornece_e_Usa" da figura 6, sendo que toda peça deve ter sido fornecida por um fornecedor para usar nos equipamentos:

Considere que as definições dos tipos de relacionamentos binários fornece e usa são:

```
DEFINE RELACIONAMENTO Fornece
  OPOSTO Fornecida
  ORIGEM Fornecedor
  DESTINO Peca
  MINIMO 1
  MINIMO OPOSTO 1;
DEFINE RELACIONAMENTO Usa
  OPOSTO Usada
  ORIGEM Equipamento
  DESTINO Peca
  MINIMO 1
  MINIMO OPOSTO 1;
```

Então o relacionamento triplo "Fornece_e_Usa" pode ser definido no esquema como:

```
DEFINE RELACIONAMENTO TRIPLO Fornece_e_Usa
  MINIMO 1
  MODALIDADE UM Usada
    CORRESPONDENTE Para_Usar_No
    MINIMO 1
  MODALIDADE DOIS Fornecida
    CORRESPONDENTE Sido_Fornecida
    DIVISAO UM Peca
    DIVISAO DOIS Fornecedor
    DIVISAO TRES Equipamento;
```

Note-se que para ser possível a existência de um relacionamento triplo envolvendo os tipos de objetos indicados, devem existir os relacionamentos binários correspondentes definidos a priori. O relacionamento triplo permite especificar na realidade uma restrição sobre os relacionamentos binários admitidos na tripla. Por exemplo, um fornecedor pode fornecer peças e serviços, mas apenas peças são fornecidas para serem usadas em equipamentos.

4.5.1.1.5 - Tipo de Atributo.

A definição de um tipo de atributo no Esquema envolve a inclusão de várias informações, assim muitas destas informações são opcionais, podendo em alguns casos assumir valores "defaults". Este comando é expresso no seguinte formato:

Sintaxe:

```

DEFINE <caracteristica> <tipo_atributo>
  [TIPO DE DADO <tipo_dado>          ] /* Cadeia  */
  [VALOR[ES] (<valor>)+]
  ASSOCIADO [A]
    [RELACIONAMENTO
      (<t_obj_origem> <modalidade> <t_obj_destino>)+]
    [OBJETO (<t_objeto>)+]
    [(TRIPLO <t_rel_triplo> <t_obj_div1>
      <t_obj_div3> (<t_obj_div3>) * ]
    [MINIMO <número>          ] /* 0      */
    [MAXIMO <número>          ] /* N      */
  [ INICIALIZACAO ] ;

```

Um mesmo tipo de atributo pode estar associado a diferentes objetos e/ou relacionamentos, sendo que as cláusulas MINIMO e MAXIMO podem ser especificadas individualmente.

O tipo de dado somente é especificado se a característica for propriedade e devem ser explicitados sempre que um tipo de atributo for definido pela primeira vez. A associação com o objeto ou relacionamento pode ser feita juntamente com a respectiva multiplicidade sempre que necessário. Quando a característica do atributo for um sinônimo e a multiplicidade for omitida, o valor "default" é 1-1.

A associação também pode ser feita após a definição do tipo de atributo. Desta forma a característica e o tipo de atributo devem ser especificados, seguidos pela cláusula "ASSOCIADO" com os respectivos elementos pré-definidos.

A cláusula "VALORES" somente pode ser especificada para a característica "PROPRIEDADE", e permite definir o conjunto de valores discretos que esse atributo pode assumir.

Exemplo: Criar o tipo de atributo "Descrição" e associá-lo ao tipo de objeto

"Projeto":

```
DEFINE COMENTARIO Descricao ASSOCIADO A OBJETO Projeto;
```

Observação: a cláusula "INICIALIZACAO" refere-se a atributos sumários utilizados associados a supertipos. Esta cláusula será apresentada no item que trata da definição de Supertipo no Esquema(4.5.1.1.7).

4.5.1.1.6 - Subtipo de Objeto.

O comando de inclusão de um novo subtipo de objeto no esquema, é definido através do seguinte formato:

Sintaxe:

```
DEFINE SUBTIPO <elemento> <Nome_do_subtipo>  
    TIPO MESTRE (<Tipo_de_obj>|  
                <t_obj_origem> <modalidade> <t_obj_destino>)  
    PREDICADO <condição_de_pertinência>  
    INICIALIZACAO  
        (<tipo_de_atributo> <valor_de_atributo>)*;
```

Funcionalmente, uma vez definido um subtipo de objeto (ou de relacionamento), este se comporta como um novo tipo de objeto (ou de relacionamento).

As cláusulas "TIPO MESTRE" e "PREDICADO" são obrigatórias, sem as quais um subtipo não pode ser definido. A cláusula "INICIALIZACAO" serve para indicar quais atributos devem ser incluídos no objeto criado quando indicado como um subtipo. Essa cláusula à princípio é opcional, porém se o predicado envolver alguma condição que não corresponda à verificação da igualdade de um tipo de atributo com um valor, então esta deverá ser necessariamente indicada. Todas as condições do predicado que correspondem à verificação de igualdade entre um tipo atributo e um valor causam automaticamente que esse valor seja colocado no atributo desse tipo quando o objeto é criado.

Por exemplo, para se definir o subtipo "Administrativo" do tipo "Funcionário", baseado no fato da propriedade "Função" do tipo de objeto "Funcionário", assumir o valor de "administrativo", define-se:

```
DEFINE SUBTIPO OBJETO Administrativo
```

```
TIPO MESTRE Fucionario
PREDICADO Funcao == "administrativo"
INICIALIZACAO
    ATRIBUTO Funcao "administrativo";
```

Nesse exemplo, indicou-se explicitamente o valor que o atributo de tipo "Função" deveria ter. Essa última indicação poderia ter sido omitida, pois a condição do predicado verifica justamente essa igualdade.

Subtipos de objetos ou de relacionamentos podem ser refinados em outros subtipos ou reagrupados com outros tipos e subtipos, formando outros subtipos. A rede de subtipos é acíclica.

Note-se que todo subtipo é apenas de um tipo de objeto ou relacionamento, mas pode ser subtipo de mais de um subtipo diferente (herança múltipla).

4.5.1.1.7 - Supertipo de Objeto.

A inclusão de um novo supertipo no Esquema é definido através do comando:

Sintaxe:

```
DEFINE SUPERTIPO <Nome_do_supertipo>
    [NA] COLONIA <Tipo_de_colônia> ;
```

Objetos podem ser semanticamente agrupados, formando Supertipos de Objetos. Supertipos, ao contrário de subtipos, caracterizam outros objetos, cujos atributos sumarizam coletivamente os objetos agrupados. Não existe supertipo de relacionamento!

Todos os objetos de um supertipo devem ser de tipos que habitam um único tipo de colônia. Supertipos são definidos em um determinado tipo de colônia, e cada supertipo pode agrupar apenas objetos de uma determinada colônia.

Por exemplo: suponha que um projeto possui um laboratório, o qual consiste em um conjunto de equipamentos alocados ao projeto. Assim pode-se definir:

```
DEFINE SUPERTIPO Laboratorio
    NA COLONIA PROJETO;
```

Os atributos de um supertipo são denominados de **Atributos Sumários**. Estes

atributos somente podem ser definidos associados a objetos de supertipo. Assim a sintaxe é dada da seguinte forma (semelhante a definição de um tipo de atributo):

Sintaxe:

```
DEFINE PROPRIEDADE <atributo>
    [TIPO DE DADO <tipo_de_dado>]
    ... /* semelhante a atributo */
INICIALIZACAO
    CONTAGEM                |
    SOMA      <tipo_atributo> |
    MINIMO    <tipo_atributo> |
    MAXIMO    <tipo_atributo> |
    MEDIA     <tipo_atributo> |
    FUNCAO    <funcao> <tipo_atributo> ;
```

Na definição de um atributo sumário a cláusula "INICIALIZACAO" é obrigatória, e determina como o valor do atributo deve ser calculado, o que é feito sempre que o valor de atributo é consultado. A cláusula "TIPO DE DADO" não precisa estar definida, mas se o for deve ser igual ao definido para o tipo de atributo indicado na cláusula "INICIALIZACAO". Se a inicialização for "CONTAGEM", o tipo de dado é "LONGO". Não pode ser definida a cláusula "MINIMO" e "MAXIMO". O tipo de atributo indicado na cláusula "INICIALIZACAO" deve ser "PROPRIEDADE".

Por exemplo, se interessar manter o custo médio dos equipamentos de um laboratório, pode-se definir:

```
DEFINE PROPRIEDADE Custo_medio
    ASSOCIADO A OBJETO Laboratorio
    INICIALIZACAO MEDIA Valor;
```

4.5.1.2 - Inclusão de elementos do Modelo na Base de Dados.

Para que um elemento possa ser incluído na base de dados, seu tipo e demais informações associadas devem estar pré-definidas no esquema.

De uma forma geral os comandos de inclusão apresentam o seguinte formato:

Sintaxe geral:

```
INCLUI [<elemento>] <argumentos> ;
```

Deve-se observar que o argumento <elemento> é opcional, pois a identificação do tipo de elemento faz parte da execução (semântica) do comando.

Também foram definidos comandos que atuam na base de dados, permitindo localizar (tornar corrente) o elemento incluído, estes comandos apresentam a seguinte estrutura:

Sintaxe geral:

```
LOCALIZA [<elemento>] <argumentos> ;
```

4.5.1.2.1 - Objetos.

Objetos são incluídos na base de dados através do seguinte comando:

Sintaxe:

```
INCLUI [OBJETO] <t_objeto> (<nome_objeto>)*  
[IDENTIFICADO [POR] RELACIONAMENTO] ;
```

Se não for indicado o caminho do objeto, a inclusão do mesmo na base de dados será feita na colônia corrente do tipo fornecido no comando. Caso o caminho seja indicado a colônia deverá estar acessível. Após este comando, o último objeto incluído tornar-se-á o corrente.

Quando o objeto é identificado por um relacionamento sua inclusão deve seguir a segunda variação deste comando indicada acima. Para isso, deve haver um relacionamento corrente do mesmo tipo, especificado na definição do tipo do objeto no esquema. Note-se que pode não ser fornecido neste comando o nome do objeto, pois a sua localização desta forma, será feita através do relacionamento. Caso seja também fornecido o seu nome, este objeto poderá ser localizado de duas maneiras, isto é, através da localização do relacionamento ou localizando objeto pelo seu nome.

Um objeto já incluído na base de dados pode ser localizado através do comando "LOCALIZA", com o seguinte formato:

Sintaxe:

```
LOCALIZA [OBJETO] <t_objeto> <nome_objeto> ;
```

Para localizar um objeto identificado por relacionamento, o relacionamento deve estar corrente. Este comando é expresso obedecendo o seguinte formato:

Sintaxe:

```
LOCALIZA [OBJETO] <t_objeto> IDENTIFICADO POR  
RELACIONAMENTO ;
```

Exemplo: Incluir os objetos "P1" e "P2" do tipo projeto:

```
INCLUI Projeto P1, P2;
```

O objeto "P2" permanece como corrente, após a inclusão de "P1" e "P2", pois foi o último a ser incluído.

Observação: este comando também pode ser utilizado para incluir um subtipo na base de dados, visto que o subtipo comporta-se como qualquer objeto de outro tipo na base de dados. Por exemplo para incluir um funcionário técnico utiliza-se o comando:

```
INCLUI Tecnico "José da Silva";
```

4.5.1.2.2 - Colônia.

Este comando permite tanto incluir como tornar uma colônia corrente na base de dados, no seguinte formato:

Sintaxe:

```
CONSIDERA [COLONIA] <t_colônia> <t_objeto>  
<nome_objeto> ;
```

A manipulação de objetos na base de dados é feita nas colônias já existentes. Para que uma colônia exista deve haver um objeto que a constrinja, portanto o primeiro objeto criado na base de dados estará sempre na colônia Global (a única colônia a existir quando a base de dados é gerada), tendo-se já definido seu tipo de objeto e o tipo colônia no esquema da base de dados. Existindo o objeto, pode-se criar a colônia efetivamente (ou abri-la, caso já tenha sido criada).

Exemplo: alocar os equipamentos "E1" e "E2" para o projeto "P1":

```
CONSIDERA PROJETO Projeto P1;  
INCLUI Equipamento E1, E2;
```

O objeto "P1" do tipo "Projeto" já deve ter sido incluído na base de dados

habitando a colônia global. Será gerada sua colônia de tipo "Projeto" e nela incluídos os objetos "E1" e "E2".

Observação: Note-se que o nome de um objeto pode ser especificado através do seu caminho de acesso. Exemplo: considere que está aberta a colônia "PROJETO" constrita por P1. Pode-se localizar o objeto Equipamento E2, especificando seu nome de duas formas:

```
LOCALIZA Equipamento /P1/E2;
```

ou

```
LOCALIZA Equipamento E2;
```

A primeira forma pode ser empregada mesmo que a colônia "PROJETO" constrita por "P1" não esteja aberta, embora necessite estar acessível. Já a segunda forma acessa o objeto chamado "E2" na colônia de tipo "PROJETO" aberta.

4.5.1.2.3 - Relacionamento Binário.

Este comando permite incluir relacionamentos entre um objeto origem e vários objetos destino, como é mostrado a seguir:

Sintaxe:

```
INCLUI [RELACIONAMENTO] <tipo_relacionamento>  
      <t_objeto> (<nome_objeto>)+ ;
```

Observação: Para se incluir ou localizar um relacionamento, deve existir previamente um objeto corrente, que será o objeto origem do relacionamento. O último relacionamento incluído permanece corrente. Para tornar um relacionamento corrente utiliza-se o comando:

```
LOCALIZA [RELACIONAMENTO] <t_relacionamento>  
      <t_objeto> <nome_objeto>;
```

Exemplo: Usando o tipo de relacionamento, já definido, de modalidade "utiliza" (figura 6), entre os objetos do tipo projeto e equipamento, incluir instâncias desse tipo de relacionamento entre os objetos "P1" e "E1", e "P1" e "E2".

```
LOCALIZA Projeto P1;  
INCLUI Utiliza Equipamento /P1/E1, /P1/E2;
```

Caso a colônia do tipo "PROJETO", constrita pelo projeto "P1" já esteja aberta, o último comando pode ser simplificado para:

```
INCLUI Utiliza Equipamento E1, E2;
```

4.5.1.2.4 - Relacionamento Triplo.

Este comando permite incluir um relacionamento triplo na base de dados, apresentando dois formatos:

Sintaxe:

```
INCLUI [TRIPLO] (<modalidade_d2>
                <t_objeto> (<nome_objeto>)+) + :
                <correspondente_d3> <t_objeto> <nome_objeto> ;

ou

INCLUI [TRIPLO] <modalidade_d3>
                <t_objeto> <nome_objeto>
                <correspondente_d2> ( :
                <t_objeto> (<nome_objeto>)+
                )+ ;
```

Para realizar a inclusão de um relacionamento triplo deve haver um objeto corrente que será considerado o objeto da divisão um.

Exemplo: Usando o relacionamento triplo "Fornece_e_Usa" da figura 6, definido anteriormente, incluir na base de dados o relacionamento entre os objetos: "Pc1", "F1", "E1" e "E2", isto é, a peça "Pc1" é fornecida pelo fornecedor "F1", e que participa tanto da composição do equipamento "E1" como do "E2".

```
LOCALIZA Peca Pc1;
INCLUI Fornecida Fornecedor F1:
        Para_Usar_No Equipamento /P1/E1, /P1/E2;
```

Para localizar um relacionamento triplo procede-se de maneira semelhante à sua inclusão na base de dados, sendo que o objeto da divisão um deve estar corrente. Este comando é definido da seguinte forma:

Sintaxe:

```

LOCALIZA [TRIPLO] (<modalidade_d2>
                  <t_objeto> (<nome_objeto>)+)+ :
                  <correspondente_d3> <t_objeto> <nome_objeto> ;
ou
LOCALIZA [TRIPLO] <modalidade_d3> <t_objeto>
                  <nome_objeto>:
                  (<correspondente_d2>
                  <t_objeto> (<nome_objeto>)+)+ ;

```

4.5.1.2.5 - Valor de Atributo.

Este comando exprime a inclusão valores de atributos em algum elemento (objeto ou relacionamento) corrente na base de dados, da seguinte forma:

Sintaxe:

```

INCLUI [ATRIBUTO] [<elemento>]
        <t_atributo> <valor> (,<valor>)* ;

```

O valor de atributo a ser incluído refere-se a algum objeto ou relacionamento corrente. Desta forma pode-se indicar a qual elemento do modelo refere-se a inclusão do atributo. Caso o elemento não esteja explicitado, será tomado o último elemento localizado ou incluído na base. O número de valores deverá estar dentro dos limites especificado nas cláusulas "MINIMO" e "MAXIMO".

Exemplo: Suponha que tanto o tipo de objeto "Peça" e o tipo de relacionamento "Fornece"\ "Fornecida" entre "Peça" e "Fornecedor", tenham sido definidos no esquema e associados ao tipo de atributo "Data", significando respectivamente a data de fabricação da peça e a data de seu fornecimento. A seguir será exemplificada a inclusão deste atributo para os diferentes elementos. Considere inicialmente os seguintes comandos:

```

LOCALIZA Peça P1;
INCLUI Fornecida Fornecedor F1;

```

Pode-se incluir o atributo data de fornecimento sem especificar o elemento ao qual a inclusão está associada, pois o relacionamento foi criado na operação anterior. Isto é feito utilizando o comando:

INCLUI ATRIBUTO Data 10/10/91;

Apesar do objeto origem do relacionamento estar corrente, para incluir o atributo data de fabricação associado ao objeto, deve-se explicitar o elemento para o qual a operação de inclusão refere-se, pois o relacionamento também está corrente, usando a variação deste comando:

INCLUI ATRIBUTO OBJETO Data 07/07/91;

Para se garantir que um atributo seja incluído no relacionamento, pode-se usar a forma:

INCLUI ATRIBUTO RELACIONAMENTO Data 10/10/91;

O comando de localização de um valor de atributo é semelhante a inclusão, como pode ser observado a seguir:

Sintaxe:

LOCALIZA [ATRIBUTO] [<elemento>] <t_atributo> [<valor>];

Caso o valor não seja indicado, localiza-se o primeiro valor desse tipo de atributo no objeto/relacionamento corrente.

4.5.1.2.6 - Subtipo.

Um subtipo de objeto ou relacionamento é incluído na base de dados através de seus respectivos comandos de inclusão. Quando um subtipo é criado na base seu tipo mestre é criado automaticamente, sendo os atributos do tipo mestre inicializados de acordo com a definição do subtipo no esquema. Note-se que todos os subtipos hierarquicamente superiores àquele criado, até o tipo original, são todos criados numa única operação.

4.5.1.2.7 - Objeto de um Supertipo.

Este comando permite incluir objetos em um supertipo, no seguinte formato:

Sintaxe:

```
INCLUI [SUPERTIPO] <nome_supertipo> <nome_obj_supertipo>  
      (TIPO <t_objeto> (<nome_objeto>+)* ;
```

O supertipo é representado em uma colônia através de um objeto, *nome_obj_supertipo*. Este objeto é criado quando o primeiro objeto da colônia é incluído no supertipo. Sendo assim, o comando pode ser utilizado novamente para incluir outros objetos da colônia ao supertipo, fornecendo os mesmos argumentos tanto para o nome do supertipo como de sua respectiva instância de objeto.

Atributos e relacionamentos de um supertipo que não sumários são herdados pelos objetos que compoem o supertipo através de **Pararelacionamentos** explicitados na Meta-base de dados. Assim, a definição de atributos e relacionamentos associados a um supertipo faz com que os objetos agrupados por cada objeto daquele supertipo sejam repassados para todos os objetos agrupados por este. Atributos sumários podem ser definidos apenas para objetos de supertipo.

O exemplo a seguir demonstra a inclusão de objetos em um supertipo.

Considerando as definições no esquema: do atributo "Valor" associado ao tipo de objeto "Equipamento", do supertipo "Laboratório" e do atributo sumário "Custo_medio"; definidos da seguinte maneira:

```
DEFINE PROPRIEDADE Valor  
      TIPO DE DADO REAL  
      ASSOCIADO A OBJETO EQUIPAMENTO;  
DEFINE SUPERTIPO Laboratorio  
      NA COLONIA PROJETO;  
DEFINE PROPRIEDADE Custo_medio  
      ASSOCIADO A OBJETO Laboratorio  
      INICIALIZACAO MEDIA Valor;
```

O próximo comando cria a instância de objeto "Circuitos Digitais" do supertipo "Laboratório"; e inclui ao supertipo as instâncias de objetos "E1", "E2" e "E3" de tipo "Equipamento". Considere que as instâncias de objetos do tipo "Equipamento" já haviam sido incluídas na base de dados.

```
SUPERTIPO Laboratorio "Circuitos Digitais"  
      TIPO Equipamento E1, E2, E3;
```

Desta forma as instâncias de objetos "E1", "E2" e "E3", herdam o atributo "Valor_medio", que será calculado toda vez que este atributo for consultado.

4.5.2 - Eliminação.

Através destes comandos, os elementos do Modelo incluídos tanto no esquema como na base de dados podem ser removidos. Sua sintaxe é semelhante aos comandos de inclusão para seus respectivos elementos, utilizando o símbolo "ELIMINA".

4.5.2.1 - Eliminação de elementos do Esquema.

Qualquer elemento já definido no esquema pode ser referenciado em um comando de eliminação, acrescentando em seguida ao comando, o símbolo "TIPO". Assim este comando apresenta o seguinte formato geral:

Sintaxe geral:

```
ELIMINA TIPO <elemento> ... ;
```

Um comando de eliminação de elementos do esquema só poderá ter sucesso desde que não haja ocorrência do elemento na base de dados. Na descrição dos comandos de eliminação no esquema, considerou-se a inexistência de nenhuma de suas ocorrências na base de dados.

4.5.2.1.1 - Tipo de Objeto.

A definição de um tipo de objeto é eliminada do Esquema utilizando o comando:

Sintaxe:

```
ELIMINA TIPO OBJETO (<tipo_objeto>)* ;
```

Para que um tipo de objeto possa ser eliminado do esquema, devem ser anteriormente eliminados os tipos de relacionamento a ele associado.

Exemplo: Eliminar o tipo de objeto "Peça":

```
ELIMINA TIPO OBJETO Peca;
```

4.5.2.1.2 - Tipo de Colônia.

O comando que permite eliminar a definição do tipo de colônia pode ser executado, desde que não reste mais nenhum tipo de objeto habitando o tipo de colônia. Este comando apresenta o seguinte formato:

Sintaxe:

```
ELIMINA TIPO COLONIA <tipo_colônia> ;
```

Exemplo: Eliminar a definição do tipo de colônia "PROJETO" do esquema:

```
ELIMINA TIPO COLONIA PROJETO;
```

4.5.2.1.3 - Tipo de Relacionamento.

Uma modalidade ou um tipo de relacionamento binário podem ser eliminados do Esquema através do comando:

Sintaxe:

```
ELIMINA TIPO RELACIONAMENTO <modalidade_relacionamento>  
[OPOSTO <modalidade_relacionamento>]  
(  
    ORIGEM <tipo_objeto>  
    DESTINO <tipo_objeto>  
) + ;
```

Este comando pode ser utilizado para eliminar vários tipos de relacionamentos associados a uma modalidade, especificando para cada um deles os tipos de objeto origem e destino. A modalidade só será removida quando o seu último tipo de relacionamento for eliminado. A especificação da modalidade de relacionamento oposto é opcional. A definição de todos os tipos de atributos do relacionamento são mantidos, mesmo que deixem de estar associados a qualquer elemento do esquema.

Exemplo: Eliminar o tipo de relacionamento "Utiliza" entre os tipos de objetos "Projeto" e "Equipamento":

```
ELIMINA TIPO RELACIONAMENTO Utiliza ORIGEM Projeto  
DESTINO Equipamento;
```

Observação: neste caso a modalidade do relacionamento também será eliminada, pois existe somente um tipo de relacionamento associado.

4.5.2.1.4 - Tipo de Relacionamento Triplo.

O comando que manipula a eliminação de um relacionamento triplo apresenta o seguinte formato:

Sintaxe:

```
ELIMINA TIPO RELACIONAMENTO TRIPLO <rel_triplo>
(
    DIVISAO UM <tipo_objeto>
    DIVISAO DOIS <tipo_objeto>
    DIVISAO TRES <tipo_objeto>
)+ ;
```

Um ou mais tipos de relacionamentos triplos podem ser eliminados através deste comando. Sua definição também será removida quando o seu último tipo de relacionamento triplo for eliminado.

Exemplo: Eliminar o relacionamento triplo "Fornece_e_Usa":

```
DEFINE RELACIONAMENTO TRIPLO Fornece_e_Usa
    DIVISAO UM Peca
    DIVISAO DOIS Fornecedor
    DIVISAO TRES Equipamento ;
```

4.5.2.1.5 - Tipo de Atributo.

A eliminação de um tipo de atributo bem como sua associação a algum objeto ou relacionamento do Esquema, é realizada utilizando o comando:

Sintaxe:

```
ELIMINA TIPO ATRIBUTO <tipo_atributo>
[    ASSOCIADO [A]
    [RELACIONAMENTO
        (<modalidade> <t_obj_origem>
        <t_obj_destino>)*]
```



```
[OBJETO (<t_objeto>)*]
[TRIPLO (<t_rel_triplo> <t_obj_div1>
        <t_obj_div2> <t_obj_div3>)*]
] ;
```

Um tipo de atributo pode ser eliminado de cada tipo de elemento individualmente, através da cláusula "ASSOCIADO A", ou pode-se eliminar sua definição, desde que não esteja associado a nenhum tipo de elemento, apenas especificando o tipo de atributo.

Exemplo: Eliminar o tipo de atributo "Descrição" do tipo de objeto "Projeto":

```
ELIMINA TIPO ATRIBUTO Descricao
        ASSOCIADO A OBJETO Projeto;
```

Observação: a associação do tipo de atributo com o tipo de objeto será eliminada, mas a sua definição não. Já no exemplo abaixo, o tipo de atributo será eliminado, considerando que não há nenhum elemento associado a este tipo de atributo:

```
ELIMINA TIPO ATRIBUTO Descricao;
```

4.5.2.1.6 - Subtipo de Objeto.

A eliminação de informações que definem um subtipo podem ser eliminadas usando o comando:

Sintaxe:

```
ELIMINA TIPO SUBTIPO <elemento> <Nome_do_subtipo>
[ INICIALIZACAO
  (ATRIBUTO
    <tipo_de_atributo> [<valor_de_atributo>])+] ;
```

Através deste comando pode-se eliminar os vários atributos de inicialização do subtipo, o valor de inicialização do atributo é opcional. Omitindo-se a cláusula "INICIALIZACAO", a definição do subtipo será eliminada.

Exemplo: Eliminar a inicialização do subtipo de objeto "Administrativo":

```
ELIMINA TIPO SUBTIPO OBJETO Administrativo
        INICIALIZACAO ATRIBUTO Cargo "administrativo";
```

4.5.2.1.7 - Supertipo de Objeto.

Informações que definem um supertipo podem ser eliminadas utilizando-se o comando:

Sintaxe:

```
ELIMINA TIPO SUPERTIPO <Nome_do_supertipo>  
[ [NA] COLONIA <tipo_de_colônia>] ;
```

Este comando permite que a eliminação da definição de um supertipo, fornecendo como argumento para o comando o nome do supertipo. A cláusula "NA COLONIA" é opcional.

Exemplo: eliminar a definição do supertipo "Laboratorio" do esquema:

```
ELIMINA TIPO SUPERTIPO Laboratorio;
```

4.5.2.2 - Eliminação de elementos da base de dados.

O comando "ELIMINA" aplicado a elementos na base de dados, permite que qualquer informação já existente na base de dados seja removida. O comando segue o mesmo padrão anterior a menos da ausência do símbolo "TIPO", denotando uma operação na base de dados. Este comando pode ser generalizado da seguinte forma:

Sintaxe Geral:

```
ELIMINA [<elemento>] ... ;
```

4.5.2.2.1 - Objetos.

A eliminação de um objeto na base de dados é executada pelo comando:

Sintaxe:

```
ELIMINA [OBJETO] <t_objeto> (<nome_objeto>)* ;  
ELIMINA [OBJETO] <t_objeto>  
IDENTIFICADO POR RELACIONAMENTO ;
```

A eliminação de um objeto resulta também na eliminação dos valores de atributos associados. Já os relacionamentos com o objeto devem ser eliminados previamente.

Para eliminar um objeto identificado por um relacionamento, o relacionamento

deve estar previamente corrente.

Para eliminar o objeto corrente, o único argumento necessário é "." ou o símbolo "CORRENTE".

Exemplo: Eliminar os "Projetos" P1 e P2:

```
ELIMINA OBJETO Projeto P1, P2;
```

4.5.2.2.2 - Colônia.

A eliminação de alguma colônia que tenha sido previamente "considerada" na base de dados, pode ser executada através do comando:

Sintaxe:

```
ELIMINA [COLONIA] <t_colônia>  
          <t_objeto> <nome_objeto> ;
```

Este comando permite eliminar uma colônia da base de dados, constrita por *t_objeto nome_objeto*. Se o objeto que constringe for corrente, este pode ser especificado por "." ou pelo símbolo "CORRENTE".

A eliminação de uma colônia da base de dados só poderá ser executada quando todos os objetos que a habitam foram eliminados. O objeto que a constringe só poderá ser eliminado se a colônia for eliminada previamente.

Exemplo: Eliminar a colônia do tipo "PROJETO", constrita pelo objeto "P1":

```
LOCALIZA Projeto P1;  
ELIMINA COLONIA PROJETO .;
```

4.5.2.2.3 - Relacionamento Binário.

Um ou mais relacionamentos binários de uma mesma modalidade podem ser eliminados da base de dados através do comando:

Sintaxe:

```
ELIMINA [RELACIONAMENTO] <tipo_relacionamento>  
          [<t_objeto> (<nome_objeto>)*] ;
```

Para se eliminar um relacionamento, deve existir previamente um objeto corrente, que será o objeto origem do relacionamento. O objeto destino do relacionamento pode ser ou não ser eliminado, dependendo do valor atribuído à cláusula "SEGURANCA" na definição do tipo de relacionamento. Os valores de atributo do relacionamento serão também eliminados.

Exemplo: Eliminar os relacionamentos "Utiliza" entre o "Projeto" "P1" e os "Equipamentos" "E1" e "E2":

```
LOCALIZA Projeto P1;  
ELIMINA Utiliza Equipamento /P1/E1 /P1/E2;
```

4.5.2.2.4 - Relacionamento Triplo.

Um relacionamento triplo corrente da base de dados, é eliminado através do comando:

Sintaxe:

```
ELIMINA TRIPLO CORRENTE  
[ (: <correspondente_triplo_d2>  
  <tipo_objeto> (<nomes_objetos>)+ ) ] ;
```

Após localizar um relacionamento triplo, este comando permite eliminá-lo ou também eliminar do triplo a associação com os objetos da divisão dois, se o símbolo ":" estiver especificado.

Exemplo: Considere o relacionamento triplo "Fornece_e_Usa" entre a "Peça" "Pc1" "Sido_Fornecida" pelo "Fornecedor" "F1", "Para_Usar_No" "Equipamento" "E1" e "E2". Para eliminar desse relacionamento triplo o fato que a "Peça" "Pc1" é "usada" no "Equipamento" "E2", os seguintes comandos podem ser utilizados:

```
LOCALIZA Peca Pc1;  
LOCALIZA Fornece_e_Usa Sido_Forneceda  
  Fornecedor F1 :  
  Para_Usar_No Equipamento /P1/E1,/P1/E2;  
ELIMINA TRIPLO CORRENTE :  
  Usada Equipamento /P1/E2;
```

Todo o relacionamento triplo será eliminado, se for usado o comando:

ELIMINA TRIPLO CORRENTE;

4.5.2.2.5 - Valor de Atributo.

Um ou mais valores de atributos podem ser removidos da base de dados, utilizando-se o comando:

Sintaxe:

```
ELIMINA [ATRIBUTO] [<elemento>] <t_atributo> [<valor>
(,<valor>)*] ;
```

Valores de atributo associados a um elemento (objeto/relacionamento) corrente, podem ser eliminados individualmente - especificando-se cada valor; ou todas as ocorrências do atributo - especificando-se somente seu tipo. A eliminação não obterá sucesso se o número valores do atributo eliminado resultar num número inferior ao mínimo de ocorrências associadas ao tipo do elemento definido no esquema.

Exemplo: Pode-se eliminar o valor do atributo "Descricao" associado ao "Projeto" "P1", através dos comandos:

```
LOCALIZA Projeto P1;
ELIMINA ATRIBUTO Descricao;
```

4.5.2.2.6 - Subtipo.

Um subtipo de objeto ou relacionamento é eliminado na base de dados através de seus respectivos comandos de eliminação na base de dados. Quando um subtipo é eliminado todos os seus subtipos inferiores são também eliminados. Note-se que se um objeto é removido de um subtipo, o objeto permanece na base de dados como sendo do subtipo superior (ou do tipo mestre).

Exemplo: remover a informação da base de dados que o funcionário "João" não é mais do subtipo "Administrativo":

```
ELIMINA Administrativo João;
```

Observação: O objeto "João" continua na base de dados, mas somente como instância do tipo "Funcionario". Para eliminar o objeto da base de dados, deve-se referenciá-lo

através do tipo mestre. O próximo comando elimina o objeto "João" da base de dados:

```
ELIMINA Funcionario João;
```

4.5.2.2.7 - Objeto de um Supertipo.

Um objeto que representa um supertipo, bem como os objetos que fazem parte do supertipo, podem ser eliminados da base de dados através do comando:

Sintaxe:

```
ELIMINA [SUPERTIPO] <nome_supertipo> <nome_obj_supertipo>  
(TIPO <t_objeto> <nome_objeto>)* ;
```

Objetos podem ser eliminados da associação com o supertipo, através de sua especificação na cláusula "TIPO". Se esta cláusula for omitida, o supertipo é eliminado da colônia corrente.

Quando um supertipo é eliminado, os objetos que o constitui não são eliminados.

Exemplo: eliminar os equipamentos "E1" e "E2" do supertipo "Laboratório" denominado de "Circuitos Digitais":

```
ELIMINA SUPERTIPO Laboratorio "Circuitos Digitais"  
TIPO Equipamento E1, E2;
```

Observação: Considerou-se que a colônia "PROJETO", constrita pelo objeto "P1", esteja aberta. Pois os objetos "Circuitos Digitais", "E1" e "E2" habitam esta colônia.

Para eliminar somente o supertipo pode ser usado o seguinte comando:

```
ELIMINA SUPERTIPO Laboratorio "Circuitos Digitais";
```

4.5.3 - Substituição.

Normalmente uma substituição pode ser executada na base de dados do MRO através de uma sequência de comandos classificados como de eliminação e inclusão. No entanto, a eliminação de algum elemento da base de dados, com objetivo de uma substituição, pode significar a perda de uma grande quantidade de outras informações que não deveriam ser eliminadas. Por isso apenas alguns elementos do Modelo foram contemplados com esta classe de instrução. Os elementos para os quais existem instruções de substituição são:

4.5.3.1 - Colônia.

Pode-se substituir o tipo de objeto que restringe uma colônia, através do comando:

Sintaxe:

```
SUBSTITUI COLONIA <tipo_de_colônia>
    [PARA CONSTRITA [POR] <tipo_de_objeto_que_constringe>]
    [PARA HABITADA [POR] (<t_objeto>)+] ;
```

Este comando permite substituir o tipo de objeto que restringe uma colônia, através da cláusula "PARA CONSTRITA POR". A cláusula "PARA HABITADA POR" permite-se substituir o tipo de colônia que habita o tipo de objeto.

4.5.3.2 - Subtipo.

Subtipos de Objetos podem ter seu predicado substituído pelo comando:

Sintaxe:

```
SUBSTITUI PREDICADO <nome_do_subtipo>
    PARA PREDICADO <nova_condição> ;
```

Note-se que com esse comando as inicializações automáticas feitas anteriormente não são eliminadas, ou seja, valores de inicialização inferidas anteriormente a partir do predicado que foi substituído foram criadas na base, e estas permanecem mesmo que a condição seja substituída. Se isso não mais deve ser feito, um comando de eliminação das inicializações anteriores deve ser solicitado explicitamente. O tipo mestre de um subtipo não pode ser substituído. As operações de inicialização podem ser substituídas por comandos de eliminação e inclusão.

O comando a seguir permite-se substituir o subtipo de algum objeto na base de dados:

Sintaxe:

```
SUBSTITUI SUBTIPO <subtipo_objeto> <nome_objeto>
    PARA <subtipo_objeto> ;
```

Exemplo: Substituir o subtipo "Tecnico" do funcionário "João", para o subtipo

"Administrativo":

```
SUBSTITUI SUBTIPO Tecnico João  
PARA Administrativo;
```

4.5.3.3 - Multiplicidade e Segurança.

A multiplicidade ou a segurança podem ser alteradas através do comando:

Sintaxe:

```
SUBSTITUI (SEGURANCA | MULTIPLICIDADE)  
(RELACIONAMENTO <modalidade>  
          <t_obj_origem> <t_obj_destino> |  
  ATRIBUTO <t_atributo> |  
  TRIPLO <t_rel_triplo> [<mod_rel_d2>])  
PARA  
  (SEGURANCA:      SIM|NAO |  
   [MINIMO:        <número>]  
   [MAXIMO:        <número>]);
```

Através deste comando pode-se substituir a definição da multiplicidade de algum elemento (relacionamento, atributo ou relacionamento triplo). Este comando também permite substituir a definição de segurança de relacionamentos (binário e triplo). Quando a substituição em um relacionamento triplo for referente a modalidade de relacionamento da divisão dois, então o argumento "mod_rel_d2" deverá ser fornecido.

4.5.4 - Estabelecimento de ambiente.

Para que diversas operações do MRO possam ser desempenhadas, é necessário que o ambiente onde essas operações irão ocorrer seja antes estabelecido. Por ambiente se entende um conjunto de elementos da base que devem estar em um estado pré-determinado, tal como a especificação de objetos e relacionamentos correntes, o conjunto de colônias que serão manipuladas, e as suas versões, uma vez que a proteção de acesso do MRO atua a nível de colônias, etc. O preparo de um ambiente leva em conta a permissão estática que o usuário tenha para acessar cada colônia, o estado dessa colônia no momento, em relação ao uso que dela estão

fazendo os outros usuários, e sobre que variante da colônia os acessos subsequentes devem ser efetuados [Calônimo_91, Ferreira_91].

Para que o usuário inicie uma sessão de acesso em uma base de dados MRO, este deve ser reconhecido pelo sistema, bem como tornar sua base de dados ativa. Caso as operações anteriores sejam bem sucedidas, será aberta uma transação. Estas operações são executadas através do seguinte comando:

Sintaxe:

```
BASE <nome_da_base_de_dados> USUARIO <nome_do_usuario>  
      CODIGO <código_do_usuario> ;
```

Para que a transação aberta seja fechada e continuar a manipular a mesma base de dados abrindo uma nova transação, utiliza-se o comando:

Sintaxe:

```
GRAVA ;
```

Caso seja necessário invalidar alguma alteração efetuada na transação aberta, e iniciar uma nova transação, utiliza-se o comando:

Sintaxe:

```
RESTAURA ;
```

Para que uma colônia seja aberta ela deve estar acessível. Colônias podem ficar acessíveis para leitura ou para leitura e escrita, através do comando:

Sintaxe:

```
REQUISITA [COLONIA] <t_colônia> <t_objeto> <objeto>  
      [PARA LER|ESCREVER] ; /* ESCREVER */
```

Exemplo: tornar todos os objetos da colônia do tipo "PROJETO", constrita pelo "Projeto" "P1" acessíveis:

```
REQUISITA PROJETO Projeto P1;
```

Colônias que não mais serão usadas podem ser liberadas através do comando:

Sintaxe:

```
LIBERA [COLONIA] <t_colônia> <t_objeto> <objeto> ;
```

Apenas colônias acessíveis podem ser abertas. Portanto, para que um comando "CONSIDERA" possa ser executado, é necessário que a colônia solicitada tenha se tornado acessível anteriormente, através de um comando "REQUISITA".

Quando um usuário inicia uma sessão de acesso com uma base de dados MRO, este terá sempre a colônia global acessível para leitura e escrita. O acesso da colônia global para escrita é permitido apenas aos usuários gerentes da base de dados. Todas as demais colônias estarão fechadas e não acessíveis, e devem ser requisitadas antes que algum acesso lhes seja permitido.

Quando o usuário finaliza uma sessão de acesso comum na base de dados MRO, todas as colônias abertas são fechadas e liberadas. A transação aberta é fechada. Estas operações são executadas através do comando:

Sintaxe:

FIM ;

O acesso à um objeto de uma colônia qualquer requer que esta esteja acessível. Assim é necessário que a colônia que restringe o objeto esteja acessível quando se quer efetuar uma operação de abertura da colônia, ou de acesso à um objeto qualquer. Note-se que o acesso a qualquer objeto de uma colônia aberta pode ser feito sempre, pois para ser aberta, a colônia já tinha que estar acessível. No entanto o acesso a um objeto restrito por uma colônia não aberta somente poderá ser feito se essa colônia estiver acessível.

Pode existir mais do que uma colônia de objetos do mesmo tipo restrita pelo mesmo objeto, porém com acesso mutuamente exclusivo. Cada uma delas é uma **Variante** da mesma colônia. Podem existir variantes com acesso permitido para leitura e escrita ou apenas para leitura. Variantes já existentes podem ser **Instanciadas** para criar outras, portanto variantes são **Instâncias de Colônias de Objetos**.

Variantes que podem sofrer alterações são chamadas **Alternativas da Colônia**. As variantes são instanciadas sempre para criar alternativas, portanto, o processo de instanciação cria variantes com acesso permitido para escrita e leitura. Isso é feito através do comando:

Sintaxe:

```
INSTANCIA [COLONIA] <tipo_colônia>  
          <tipo_objeto_constringe> <objeto_constringe>  
          PARA <nome_da_variante> ;
```

O usuário pode acessar em um dado instante apenas uma variante de cada colônia, a qual é chamada de **Variante Corrente**. Para tornar corrente uma variante usa-se o comando:

Sintaxe:

```
VARIANTE [COLONIA] <tipo_colônia>  
          <tipo_objeto_constringe> <objeto_constringe>  
PARA <nome_da_variante> ;
```

Uma vez que uma variante é indicada como a corrente, esta permanece corrente para esse usuário mesmo através de sessões subsequentes, até que uma outra a substitua. Isso significa que a indicação de variantes como correntes transcende a acessibilidade das mesmas, apesar de ser necessário que se tenha o objeto que restringe uma colônia acessível para que se possa indicar qual de suas variantes é a corrente. No entanto, para que uma variante corrente possa ser mudada, é necessário que a própria colônia não esteja acessível.

Para que uma variante seja instanciada, esta deve estar acessível, e após a instanciação, o usuário passa a ter acesso à nova alternativa criada. Para voltar a ter acesso à variante original é necessário utilizar-se o comando "VARIANTE".

Uma vez que uma Alternativa esteja pronta, esta pode ser **Congelada**, transformando-se em uma **Versão da Colônia**. Isso é feito através do comando:

Sintaxe:

```
CONGELA [COLONIA] <tipo_colônia>  
          <t_objeto> <objeto_que_constringe> ;
```

Para que uma variante seja congelada, esta não pode estar acessível. A variante que é congelada é a corrente.

4.5.5 - Seleção de objetos.

Objetos da base de dados podem ser agrupados, constituindo-se assim em **Seleções de objetos**. Os objetos de uma seleção podem habitar diferentes colônias ou tipos de colônias. Uma seleção pode ser:

Transiente: é a seleção (única) que existe somente durante o momento em que está sendo manipulada, desaparecendo após deixar de ser corrente. Esta

seleção pode ser referenciada através do símbolo "@";

Persistente: são seleções previamente e explicitamente definidas, podendo existir diversas seleções persistentes. Uma seleção persistente sempre tem um nome designado pelo usuário, e permanece na base de dados até que seja explicitamente eliminada pelo usuário.

Os comandos que manipulam seleções podem ser subdivididos em:

- Gerenciamento de seleção;
- Obtenção de seleção a partir de outras já existentes;
- Operações válidas para seleção;
- Acesso a cada objeto de uma seleção.

4.5.5.1 - Gerenciamento de seleção.

Estes comandos permitem fazer as manipulações básicas em seleções, tais como: criação, inicialização, eliminação, etc.:

i) A inicialização de seleção é executada pelo comando:

Sintaxe:

```
DEFINE SELECAO [<nome_seleção>] ;
```

Este comando inicializa uma nova seleção que será a corrente. No caso do identificador estar especificado, a seleção será persistente, caso contrário, será transiente.

ii) Pode-se tornar corrente uma seleção que já foi previamente definida, através do comando:

Sintaxe:

```
LOCALIZA SELECAO <nome_seleção> ;
```

iii) Seleções persistentes podem ser eliminadas da base de dados através do comando:

Sintaxe:

```
ELIMINA SELECAO <nome_seleção> ;
```

- iv) Tendo uma seleção corrente, pode-se incluir nela os objetos de uma outra seleção através da instrução:

Sintaxe:

ADICIONA <seleção> ;

- v) Para a seleção corrente pode-se excluir objetos que dela participam através do comando:

Sintaxe:

EXCLUI <t_objeto> (<nome_objeto>)+ ;

4.5.5.2 - Obtenção de seleção a partir de outras já existentes.

A seguir será descrito como pode-se especificar através dos comandos da linguagem obter estas seleções, a partir das já existentes na base de dados, tais como: colônias, tipos de objetos, supertipos, etc. Desta forma uma seleção pode ser:

- i) Objetos de um mesmo tipo isolados:

Sintaxe:

<t_objeto> (<nome_objeto>)*

Pode-se referenciar um objeto corrente através do símbolo: ".".

- ii) Objetos de um determinado tipo que habitam uma colônia aberta:

Sintaxe:

TIPO <t_objeto>

- iii) Objetos que habitam uma colônia correntemente aberta:

Sintaxe:

HABITAM <tipo_de_colônia>

- iv) Objetos de algum supertipo na colônia corrente:

Sintaxe:

SUPERTIPO <nome_supertipo> <nome_objeto>

v) Objetos de alguma seleção persistente, isto é, já definida:

Sintaxe:

SELECAO <nome_seleção>

Exemplo: Obter todos os objetos disponíveis do tipo "Peça" e todos os objetos da colônia "PROJETO" que é constricta pelo "Projeto" "P1":

```
DEFINE SELECAO;  
ADICIONA TIPO Peca;  
REQUISITA PROJETO Projeto P1  
      PARA LER; /* Torna a colônia acessível */  
CONSIDERA PROJETO Projeto P1; /* torna a colônia  
                                disponível */  
ADICIONA HABITAM PROJETO; /* Inclui os objetos  
                            disponíveis da colônia  
                            PROJETO */
```

Observação: aqui considerou-se que todos os objetos do tipo "Peça" habitam uma colônia disponível, que não é a colônia "PROJETO", nem subordinada a ela.

4.5.5.3 - Operações válidas em seleção.

As operações definidas para os objetos de seleção são:

- Operações de conjuntos;
- Operações envolvendo relacionamento;
- Operações envolvendo atributos.

A finalidade destas operações é tornar poderosa a manipulação de seleções, reunindo operações de diferentes conceitos de operações da álgebra de conjuntos e operações permitidas pelas características do modelo tais como relacionamento e atributo.

4.5.5.3.1 - Operações de conjuntos.

Os operadores tradicionais de conjuntos, tais como: União, Intersecção e Diferença, estão disponíveis, através dos operadores: "+", "^" e "-", respectivamente. Estes operadores

consideram que uma seleção é um conjunto de objetos, observando suas propriedades de conjunto, portanto sem repetições e sem importar a ordem dos objetos na seleção. Estas operações resultam em uma nova seleção.

<seleção1> <op_conjunto> <seleção2>

4.5.5.3.2 - Operações envolvendo Relacionamento.

Através destas operações de seleção, permite-se que os relacionamentos incluídos na base de dados, possam ser utilizados para restringir a inclusão de objetos nas seleções.

- i) Para obter uma seleção de objetos que fazem parte da origem do relacionamento:

Sintaxe geral:

<seleção1> <relacionamento> <seleção2>

O resultado dessa seleção são os objetos da seleção 1 que tenham esse tipo de relacionamento com algum dos objetos da seleção 2.

Exemplo: Obter a seleção que contenha somente as peças que têm algum fornecedor.

```
DEFINE SELECAO Peca_fornecida;  
ADICIONA TIPO Peca Fornecida TIPO Projeto;
```

Observação: Somente as peças que tenham relacionamento com algum fornecedor serão selecionadas.

- ii) Pode-se obter os objetos que fazem parte de um relacionamento sem identificar-se a seleção de objetos que é destino:

Sintaxe geral:

<seleção> -\$ <relacionamento>

O símbolo "-\$" pode ser interpretado como "que tem".

Exemplo: Obter todos os objetos do tipo "Peça" que têm relacionamento de modalidade "Fornecida":

```
DEFINA SELECAO Peca_fornecida;  
ADICIONA TIPO Peca -$ Fornecida;
```

- iii) O comportamento da operação na seleção pode ser alterado, para permitir obter objetos (seleção) da seleção destino:

Sintaxe geral:

<selecao1> DESTINO <relacionamento> <selecao2>

ou

<selecao1> DESTINO -\$ <relacionamento>

O resultado dessa seleção são os objetos que participam de algum relacionamento desse tipo, cuja origem seja um objeto da seleção 1.

Exemplo: Obter objetos do tipo equipamento constrito pelo projeto P1, a partir da seleção de "Peca_fornecida".

```
DEFINE SELECAO;  
Peca_fornecida DESTINO Usa  
TIPO Equipamento CONSTRITO POR P1;
```

- iv) Os objetos obtidos nas operações com relacionamento anteriores são aqueles que possuem relacionamento da modalidade indicada. Para obter-se o seu complemento, isto é, os objetos que não possuem a modalidade de relacionamento, utiliza-se o operador de negação "~", antes do nome da modalidade do relacionamento ou antes do símbolo "-\$".

Exemplo: Obter as peças que não são fornecidas pelo fornecedor "F1":

```
DEFINE SELECAO;  
ADICIONA TIPO Peca ~Fornecida Fornecedor F1;
```

4.5.5.3.3 - Operações envolvendo Atributo.

Aqui a seleção de objetos resultante será o conjunto de objetos da seleção original que satisfaça o predicado envolvendo atributos e valores indicados. Apresentando o formato:

Sintaxe Geral:

<seleção> <predicado>

As comparações que podem fazer parte do predicado são: ==, !=, <, <=, > e >=,

podendo ser compostas por conectivos OR, AND e NOT. Somente são válidas as comparações envolvendo valores de atributo que tenham característica de **propriedade** ou **sinônimo**.

Exemplo: Obter os projetos localizados na cidade de São Carlos.

```
DEFINE SELECAO Projeto_SC;  
ADIOCINA TIPO Projeto Cidade == "Sao Carlos";
```

Como na restrição por relacionamento, uma seleção também pode ser reagrupada em somente objetos que tenham ("-\$\$") um determinado atributo, sem especificar o seu valor, no seguinte formato:

Sintaxe geral:

```
<seleção> -$ <atributo>
```

Observação: Aqui também pode ser usado o operador de complemento "~" antes do símbolo "\$".

4.5.5.4 - Acesso a cada objeto de uma seleção.

Cada objeto da seleção pode tornar-se corrente e manipulado por qualquer outro comando da linguagem, através do comando:

Sintaxe geral:

```
LOCALIZA CADA <seleção> {  
    /* um objeto da seleção torna-se corrente */  
}
```

Exemplo: Mudar as cidades dos projetos que estão alocados em São Carlos para São Paulo.

```
LOCALIZA SELECAO Projeto_SC;  
LOCALIZA CADA @ {  
    ELIMINA ATRIBUTO Cidade;  
    INCLUI ATRIBUTO Cidade "Sao Paulo";
```

Observação: o símbolo "@" é utilizado para referenciar a seleção corrente (no caso a seleção "Projeto_SC").

4.5.6 - Lista/Máscara para seleções.

O comando "LISTA" permite listar o nome dos objetos que participam de uma seleção. Seu formato geral é:

Sintaxe geral:

```
LISTA <seleção> ;
```

Máscara pode ser entendida como um filtro aplicado sobre a seleção.

Máscara define um conjunto de tipos de atributos e modalidades de relacionamentos, que já foram definidos previamente no esquema. Pode-se criar uma máscara, onde o conjunto de tipos de atributos e relacionamentos, bem como sua ordem, são indicados. Para isso utiliza-se o comando:

Sintaxe:

```
DEFINE MASCARA <nome> '{'
    (<t_atributo>|<m_relacionamento>)*
    '}'
    [ORDEM CRESCENTE|DECRESCENTE
      (<t_atributo>|<m_relacionamento>)]
    [REPETICAO SIM|NAO] ; /* assume SIM */
```

Uma máscara permite que se manipulem as informações relativas a objetos, ou relacionamentos, cujos atributos deseja-se indicar.

Na definição de uma máscara admite opcionalmente a especificação de ordem e/ou repetição, com que os dados mascarados podem ser utilizados.

Máscaras são sempre utilizadas juntamente com seleções, criando-se tabelas de objetos, onde os atributos da máscara constituem os atributos e os relacionamentos, aos quais se reporta a operação em andamento.

Para a utilização das máscaras, já definidas ou não, deve-se acrescentar a cláusula "MASCARA", do comando "LISTA", no seguinte formato:

Sintaxe:

```
LISTA <selecao> MASCARA <nome> ;
ou
LISTA <selecao> MASCARA '{'
      (<atributo> | <relacionamento>)* '}'
      [ORDEM CRESCENTE|DECRESCENTE
      (<t_atributo>|<m_relacionamento>)]
      [REPETICAO SIM|NAO] ; /* assume SIM */
```

Para manter a generalidade da máscara no MRO, mesmo frente a objetos de diversos tipos na seleção, os quais a princípio possuem atributos e relacionamentos diversificados que não poderiam ser agrupados em uma mesma máscara. Desta forma as seguintes regras de integridade são aplicadas às máscaras:

1. Atributos da máscara que não estão presentes no objeto, embora definidos no tipo de objeto, terão valor "INDEFINIDO" na tupla;
2. Atributos da máscara que não podem ser associados ao objeto terão valor "INVÁLIDO" na tupla;
3. Atributos do objeto que não estão na máscara são simplesmente ignorados;
4. A atribuição de um valor na tupla para um atributo que pode ser associado ao objeto, definirá o atributo para o objeto;
5. A definição de um valor de atributo na tupla que não pode ser associado a um objeto, será ignorado na operação do gerenciador da base MRO, e representa um erro para o interpretador da linguagem de acesso.

Uma máscara é definida como um conjunto de atributos, aplicáveis indistintamente a objetos ou a relacionamentos. Os atributos podem ser **Intrínsecos** ou **Extrínsecos**. Os atributos intrínsecos são aqueles definidos pelo próprio modelo, sendo invariantes para toda aplicação. Os atributos Extrínsecos são aqueles definidos pela modelagem feita para a aplicação.

Os atributos intrínsecos são indicados através de símbolos definidos na tabela apresentada a seguir.

Quando a máscara é aplicada a um objeto:

Símbolo:

α NOME

Atributo:

Identificador principal do objeto;

αCODIGO	Código do objeto;
αTIPO	Tipo do objeto;
αCONTEXTO	Contexto de colônias para acesso ao objeto;
αSTATUS	Situação do objeto: disponível, bloqueado,...;
αCRIADO	Data de criação do objeto;
αMODIFICADO	Data da última atualização do objeto;
αDONO	Usuário que criou o objeto;
αATUALIZADOR	Usuário que modificou o objeto pela última vez.

Quando a máscara é aplicada a um relacionamento:

Símbolo:	Atributo:
αRELACIONAMENTO	Modalidade de relacionamento associada ao objeto;
αOPOSTO	Modalidade de relacionamento oposto associada ao objeto;
αMULMINIMA	Multiplicidade mínima do tipo de relacionamento;
αMULMAXIMA	Multiplicidade máxima do tipo de relacionamento.

Os atributos extrínsecos, no contexto desta linguagem, referem-se:

- Aos atributos associados a objetos do tipo em questão;
- Ao nome dos objetos destino dos relacionamentos associados ao objeto em questão;
- Aos atributos associados à esses relacionamentos;
- E aos atributos associados aos objetos destino dos relacionamentos.

Observações:

- a) Para o casos de relacionamentos binários que tenham mais que um objeto destino e atributos multivalorados; este comando apresenta uma restrição, isto é, somente permite listar o primeiro nome de objeto destino ou o primeiro valor do atributo associado ao objeto listado na seleção;
- b) Os atributos de um relacionamento são descritos na especificação de uma máscara utilizando uma sintaxe especial:

Sintaxe:

<m_relacionamento>'.'<t_atributo>

Exemplos:

i)Listar a seleção de peças, segundo a máscara que define o nome das peças, o atributo "Data_Fornecimento" e nome do objeto destino associados ao relacionamento de modalidade "Fornecida":

```
DEFINE MASCARA Tabela1
      {αNOME, Fornecida.Data_Fornecimento, Fornecida};
LISTA TIPO Peca MASCARA Tabela1;
```

ii)Listar os projetos que iniciaram a partir de 15/03/90:

```
LISTA (TIPO Projeto Data_inicio >= 15/03/90)
      MASCARA {αNOME, Data_inicio};
```

iii)Listar os funcionários que participam do projeto tomógrafo:

```
LISTA ((TIPO Funcionario)
      Participa
      (TIPO Projeto αNOME == "Tomografo"))
      MASCARA {αNOME,cic,Data_admissao};
```

4.5.7 - Mostra elemento.

O comando "MOSTRA" foi criado com o objetivo de permitir ao usuário realizar uma consulta rápida sobre o estado em que a base encontra-se um elemento num determinado momento. Desta forma o comando deve ser versátil o suficiente para possibilitar visualizar a base segundo seus diferentes elementos, tais como: colônia, tipo, objeto, relacionamento, atributo, supertipo, subtipo, etc., ao mesmo tempo deve ser simples de utilizar.

A utilização deste comando não afeta o estado corrente da base, sendo que o elemento a ser observado não é necessariamente o elemento pertencente ao contexto corrente.

A abordagem utilizada foi a de manter rigidez na definição das informações dos elementos a serem exibidas, sendo que as informações do elemento consultado são pré-definidas. A flexibilidade na escolha das informações exibidas é uma característica do comando "LISTA" associado ao comando "MASCARA" os quais são utilizados para exibir objetos agrupados por

alguma operação de seleção.

Uma característica do comando é permitir que uma mesma informação possa ser obtida segundo diferentes níveis de detalhe. Por exemplo, um mesmo objeto pode ser listado pelo comando que mostra os objetos que com ele habitam alguma colônia ou utilizando o comando que mostra um objeto. Neste caso outras informações serão exibidas tais como os valores de seus atributos, relacionamentos, etc.

A seguir será descrito o comportamento do comando mostra para cada um dos elementos do Modelo, sendo apresentado o comando "MOSTRA" para o esquema e a base de dados.

4.5.7.1 - Mostra elementos do esquema.

4.5.7.1.1 - Tipo de Objeto.

i) Os tipos de objetos definidos no Esquema, podem ser mostrados através do comando:

Sintaxe:

```
MOSTRA TIPOS OBJETOS ;
```

São mostrados os tipos de objetos definidos no esquema, sendo que para cada um são exibidas as seguintes informações:

- Tipo de objeto;
- O tipo de colônia a qual ele pertence.
- O tipos de relacionamentos para o qual é origem;
- O tipos de atributos associados.

Exemplo: Mostrar os tipos de objetos definidos no esquema:

```
MOSTRA TIPOS OBJETOS;
```

Resulta em:

```
Nome = Peca;  
      habita global;
```

```
Nome = Fornecedor;  
      habita global;  
Nome = Projeto  
      habita global;  
Nome = Funcionario;  
      habita global;  
Nome = Equipamento;  
      habita PROJETO;
```

ii) Um tipo de objeto, de interesse específico, pode ser mostrado através do comando:

Sintaxe:

```
MOSTRA TIPO OBJETO <tipo_objeto>;
```

Neste comando são obtidas as mesmas informações do comando anterior, mas só somente para o tipo de objeto discriminado.

4.5.7.1.2 - Tipo de Colônia.

i) Todos os tipos de colônias definidas no Esquema, podem ser mostradas usando o comando:

Sintaxe:

```
MOSTRA TIPOS COLONIAS ;
```

As informações de cada colônia exibidas por este comando são:

- Nome da colônia;
- Tipo de objeto que a restringe;
- Tipo de objetos que habitam o tipo de colônia;
- Índice da colônia.

Observação: A toda colônia é associado um índice, sendo que para colônia global seu valor é 0.

ii) Pode-se obter informações a respeito da definição de somente um tipo de colônia, através do uso do comando:

Sintaxe:

MOSTRA TIPO COLONIA <tipo_colônia> ;

As informações exibidas por este comando são:

- Nome da colônia;
- Tipo de objeto que a constri;
- Tipos de objetos que habitam o tipo de colônia.

Exemplo: mostrar todos os tipos de colônias definidas no esquema da empresa de engenharia:

MOSTRA TIPOS COLONIAS;

Resulta em:

```
Nome = global;  
      indice = 0;  
      habitada Fornecedor, Peca, Projeto, Funcionario;  
Nome = PROJETO;  
      indice = 3;  
      constrita Projeto;  
      habitada Equipamento;
```

4.5.7.1.3 - Tipo de Relacionamento Binário.

i)As modalidades e os tipos de relacionamentos binários associados definidos no Esquema, podem ser mostrados, através do comando:

Sintaxe:

MOSTRA TIPOS RELACIONAMENTOS ;

Este comando exibe os tipos de relacionamentos existentes para cada modalidade de relacionamento, mostrando as seguintes informações:

- Nome da modalidade de relacionamento;
- Nome da modalidade de relacionamento oposto;
- Tipo de objeto origem;
- Tipo de objeto destino.

Exemplo: Mostrar todos os relacionamentos binários definidos no esquema:

MOSTRA TIPOS RELACIONAMENTOS;

Resulta em:

```
Nome = Fornece;  
      oposto Fornecida;  
      origem Fornecedor;  
      destino Peca;  
Nome = Usa;  
      oposto Usada;  
      origem Peca;  
      destino Equipamento;
```

ii) Pode-se obter os tipos de relacionamento associados a uma modalidade específica, através do comando:

Sintaxe:

MOSTRA TIPO RELACIONAMENTO <modalidade>

;

Para uma modalidade de relacionamento este comando mostra a modalidade de relacionamento oposta, seguido das seguintes informações para cada tipo de relacionamento binário de que participa:

- Tipo de objeto origem;
- Opção por segurança ("SIM|NAO");
- Multiplicidade mínima;
- Multiplicidade máxima;
- Tipo de objeto destino;
- Multiplicidade mínima oposta;
- Multiplicidade máxima oposta;
- Tipos de atributos associados.

4.5.7.1.4 - Tipo de Relacionamento Triplo.

i) Os tipos de relacionamentos triplos podem ser mostrados, através do comando:

Sintaxe:

MOSTRA TIPOS RELACIONAMENTOS TRIPLOS ;

Este comando mostra as seguintes informações para cada tipo de relacionamento triplo:

- Nome do tipo de relacionamento triplo;
- Nome da modalidade binária da divisão dois;
- Nome da modalidade binária da divisão três;
- Nome do tipo de objeto pertencente a divisão um;
- Nome dos tipos de objetos pertencentes a divisão dois;
- Nome do tipo de objeto pertencente a divisão três.

ii) Um tipo de relacionamento triplo pode ser mostrado através do comando:

Sintaxe:

MOSTRA TIPO RELACIONAMENTO TRIPLO <relacionamento>;

Este comando permite consultar todos os tipos de relacionamentos triplos existentes no esquema sendo manipulado, e para cada um, mostra as seguintes informações:

- Nome do tipo de relacionamento triplo;
- Multiplicidade máxima do relacionamento triplo;
- Multiplicidade mínima do relacionamento triplo;
- Opção por segurança ("SIM|NAO").
- Nome da modalidade que compõe a a divisão dois;
- Seu nome correspondente atribuído na definição do triplo;
- Multiplicidade máxima do relacionamento da divisão;
- Multiplicidade mínima do relacionamento da divisão;
- Nome da modalidade que compõe a divisão três;
- Seu nome correspondente atribuído na definição do triplo;
- Nome do tipo de objeto pertencente a divisão um;
- Nome do tipo de objeto pertencente a divisão dois;

4.5.7.1.5 - Tipo de Atributo.

i) Todos os tipos de atributos definidos no Esquema, podem ser mostrados utilizando-se o comando:

Sintaxe:

```
MOSTRA TIPOS ATRIBUTOS ;
```

Para este comando mostra, pode-se obter as seguintes dos tipos de atributos definidos:

- Nome do atributo;
- Característica;
- Tipo de Dado;

ii) Somente um tipo de atributo definido no Esquema pode ser mostrado através do comando:

Sintaxe:

```
MOSTRA TIPO ATRIBUTO <tipo_atributo> ;
```

As informações obtidas são:

- Nome do atributo;
- Característica;
- Tipo de Dado;
- Multiplicidade mínima;
- Multiplicidade máxima;
- Nome de tipos de objetos ou relacionamentos associados.

Exemplo: Mostrar o tipo de atributo "Data_de_Inicio":

```
MOSTRA TIPO ATRIBUTO CIC;
```

Serão mostradas as seguintes informações:

```
Nome = CIC;
      característica sinonimo (1);
      tipo_de_dado cadeia (7);
atributo_de Funcionario;
      minimo 1;
      maximo 1;
```

4.5.7.1.6 - Subtipos.

Os subtipos definidos podem ser mostrados como tipos de objetos, porém as informações mostradas não permitirão distingui-los, assim o comando **mostra** para subtipo permite observar informações específicas da definição do subtipo.

i) Os subtipos definidos tanto de objetos como de relacionamentos, podem ser mostrados através do comando:

Sintaxe:

```
MOSTRA TIPOS SUBTIPOS ;
```

Através deste comando, pode-se obter as seguintes informações para cada subtipo definido no esquema:

- Nome do subtipo;
- Tipo de objeto/relacionamento mestre do subtipo.

Exemplo: Mostrar os subtipos definidos no esquema:

```
MOSTRA SUBTIPOS;
```

Resulta em:

```
Nome = Pessoa --> Administrativo;
```

```
Nome = Pessoa --> Técnico;
```

ii) A definição de somente um subtipo no Esquema, é mostrada através do comando:

Sintaxe:

```
MOSTRA SUBTIPO OBJETO <nome_subtipo> ;
```

Este comando mostra as seguintes informações de um subtipo:

- Nome do subtipo;
- Tipo de objeto mestre do subtipo;
- Tipos de atributos associados ao subtipo;
- Predicado que determina a condição para um objeto do tipo mestre pertencer também ao subtipo;

- Inicialização que os atributos dos objetos do subtipo recebem quando são definidos.

4.5.7.1.7 - Supertipo.

i) Todos os supertipos definidos no Esquema podem ser mostrados através do comando:

Sintaxe:

```
MOSTRA TIPOS SUPERTIPOS ;
```

Este comando mostra todos os supertipos definidos no esquema da aplicação, sendo que para cada supertipo serão exibidas as seguintes informações:

- Nome do supertipo;
- Tipo de colônia na qual este supertipo está definido.

ii) Mostra um supertipo específico.

Sintaxe:

```
MOSTRA TIPO SUPERTIPO <nome_supertipo> ;
```

Este comando mostra um supertipo de interesse específico, exibindo as seguintes informações:

- Nome do supertipo;
- Tipo de colônia na qual este supertipo está definido.
- Tipos de atributos associados a este supertipo.

4.5.7.2 - Mostra elementos da Base de Dados.

4.5.7.2.1 - Objeto.

i) Um objeto que habita a colônia correntemente aberta, pode ser mostrado através do comando:

Sintaxe:

MOSTRA [OBJETO] <tipo_objeto> <nome_objeto> ;

Neste comando são exibidas as seguintes informações:

- Nome e tipo do objeto;
- Contexto em relação a hierarquia de colônias;
- Modalidade de relacionamento e objeto destino associados;
- Tipos e valores de atributos associados.

Exemplo: mostrar o equipamento "E1":

MOSTRA Equipamento P1/E1;

Resulta em:

Nome = (Equipamento) P1/E1
Utilizado Projeto P1,
Projeto P2;
Valor 100000;

ii) Todos os objetos que habitam alguma colônia aberta ou acessível, podem ser mostrados através do comando:

Sintaxe:

MOSTRA [COLONIA] <tipo_colonia> ;

Neste comando são mostradas as seguintes informações sobre os objetos:

- Nome principal do objeto;
- Tipo do objeto.

4.5.7.2.2 - Colônia.

i) Todas as colônias que foram abertas, são mostradas usando-se o seguinte comando:

Sintaxe:

MOSTRA COLONIAS ABERTAS ;

É mostrada a sequência de colônias abertas: sendo que para cada uma delas são

mostradas as seguintes informações:

- Nome da colônia;
- Tipo de acesso permitido.

Exemplo: Considere que a colônia "PROJETO" constrita pelo projeto P1 é a corrente. Mostrar as colônias abertas:

```
MOSTRA COLONIAS ABERTAS;
```

Resultando em:

```
Nome = global;
```

```
Tipo de acesso: LEITURA;
```

ii) Pode-se obter as informações de todas as colônias que estão acessíveis, através do comando:

Sintaxe:

```
MOSTRA COLONIAS ACESSIVEIS ;
```

Este comando mostra todas as colônias para as quais foi solicitado e obtido acesso, e ainda não liberadas, sendo que para cada colônia mostra-se:

- Nome da colônia;
- Tipo de acesso permitido.

Observação: O comando mostra apresenta uma diversidade de funcionalidades tanto para objetos como para colônias na base de dados. A finalidade disto é permitir ao usuário situar o seu contexto, bem como auxiliar na "navegação" na base de dados. Para os demais elementos o comando **mostra** apresenta o seguinte formato:

Sintaxe geral:

```
MOSTRA [<elemento>] <argumentos> ;
```

4.5.7.2.3 - Relacionamento Binário.

O comando que permite exibir as informações de um relacionamento na base de dados, apresenta o seguinte formato:

Sintaxe:

MOSTRA RELACIONAMENTO CORRENTE ;

A exigência deste comando é que deve existir um relacionamento corrente para que este seja mostrado. As informações exibidas são:

- Modalidade do relacionamento, tipo e nome do objeto origem e destino;
- Tipo e valor de atributos associados ao relacionamento.

Exemplo: mostrar o relacionamento de modalidade "Utiliza" entre o projeto "P1" e o equipamento "E1":

```
LOCALIZA Projeto P1;  
LOCALIZA Utiliza Equipamento /P1/E1;  
MOSTRA RELACIONAMENTO CORRENTE;
```

Resultando em:

```
Nome = P1;  
      Utiliza E1;
```

Observação: os comandos "LOCALIZA" do exemplo anterior, foram necessários para tornar corrente o relacionamento alvo.

4.5.7.2.4 - Relacionamento Triplo.

As informações de um relacionamento triplo incluído na base de dados podem ser obtidas através do comando:

Sintaxe:

MOSTRA TRIPLO CORRENTE ;

Para um relacionamento triplo corrente da base este comando mostra:

- Nome e tipo do objeto da divisão 1;
- Modalidade do relacionamento da divisão 2;
- Nome e tipo de todos os objetos da divisão 2;
- Modalidade do relacionamento da divisão 3;
- Nome e tipo do objeto da divisão 3;

- Atributos do relacionamento triplo (tipo e valor).

4.5.7.2.5 - Valor de Atributo.

Os valores de um atributo em específico que está associado ao elemento corrente, podem ser obtidos através do comando:

Sintaxe:

```
MOSTRA [ATRIBUTO] [<elemento>] <tipo_atributo> ;
```

Este comando mostra informações detalhadas de um valor de atributo, para um objeto ou relacionamento corrente, as informações exibidas são:

- Valores associados ao tipo de atributo para o elemento corrente.

Exemplo: Mostrar o atributo tipo "Valor" do equipamento "E1":

```
LOCALIZA Equipamento /P1/E1;
```

```
MOSTRA ATRIBUTO Valor;
```

Resultando em:

```
Valor = 10000;
```

4.5.7.2.6 - Subtipo.

As informações do subtipo de objeto podem ser mostradas utilizando o comando "MOSTRA" para um objeto ou relacionamento.

Como um subtipo é essencialmente um tipo, pode-se utilizar os comandos referentes a objetos, substituindo o nome do tipo de objeto, pelo nome do subtipo. Todas as informações do subtipo em questão são dadas como se fossem um objeto.

4.5.7.2.7 - Supertipo.

As informações de um supertipo que habita a colônia correntemente aberta, podem ser obtidas através do comando:

Sintaxe:

MOSTRA SUPERTIPO <nome_supertipo> <nome_objeto> ;

Serão exibidas as seguintes informações:

- Tipo e valor dos atributos pertencentes ao supertipo;
- Tipo e nome dos objetos pertencentes ao supertipo.

4.6 - Considerações sobre a concepção dos comandos.

A seguir são feitas algumas considerações a respeito de como alguns comandos foram concebidos. Analisando tanto a sintaxe como a semântica dos comandos definidas nesta linguagem.

Os comandos de manipulação do esquema, foram construídos principalmente a partir da observação do meta-esquema do MRO, representado através do seu DRO da figura do apêndice 2. Nele representam-se o comportamento e a interação dos elementos do Modelo, a partir do qual pode-se extrair o comportamento sintático e semântico dos elementos.

Por exemplo, considere a sintaxe do comando que define uma colônia no esquema:

Sintaxe:

```
DEFINE COLONIA <tipo_colônia>  
    CONSTRITA POR <tipo_objeto>  
    HABITADA POR (<tipo_objeto>)+ ;
```

Segue-se abaixo a descrição da sintaxe e da semântica respectivamente do comando.

DEFINE: identifica uma operação de inclusão no esquema;

COLONIA: identifica o elemento do modelo manipulado;

<tipo_colônia>: nome principal do objeto do tipo "colônia" a ser criado ou localizado na colônia "βEsquema";

CONSTRITA POR: modalidade do tipo de relacionamento entre os tipos de objetos "colônia" e "objeto";

<tipo_objeto>: especifica o nome do objeto de tipo "objeto", o qual será destino do relacionamento de modalidade "Habitada por" com origem no objeto de

nome "tipo_colônia" de tipo "colônia". O objeto "<tipo_objeto>" já deve ter sido previamente criado por um comando que define um tipo de objeto. A multiplicidade do tipo de relacionamento é de 1-1, significando que deve haver no mínimo e no máximo um relacionamento desta modalidade com origem no objeto "<tipo_colônia>". Isto explica a necessidade da obrigatoriedade de fornecer pelo menos um argumento na cláusula "CONSTRITA POR".

HABITADA POR: modalidade do tipo de relacionamento entre os tipos de objetos "colônia" e "objeto";

(<tipo_objeto>)+: especifica os objetos destino dos relacionamentos com origem no objeto "<tipo_colônia>" e modalidade "habitada_por". Caso os objetos destino já tenham sido previamente através do comando que define um tipo de objeto, necessita-se apenas incluir os relacionamentos de modalidade "habitada_por", caso contrário antes de se incluir o relacionamento, deve-se incluir o objeto "<tipo_objeto>". A multiplicidade do relacionamento é de 1-N, indicando que deve haver no mínimo um relacionamento criado, isso faz com que seja necessário que seja fornecido pelo menos um "tipo_objeto".

Assim este comando cria um objeto do tipo "colônia", fornecido pelo argumento do comando "<tipo_colônia>". Cria os relacionamentos de modalidade "Constrita por" entre os com origem no objeto "<tipo_colônia>" e destino em "tipo objeto", fornecido após a cláusula "CONSTRITO POR". Cria os relacionamentos de modalidade "habitada_por" com origem no objeto "tipo_colônia" e destino fornecido pelos argumentos do comando "(<tipo_objeto>)*".

Em alguns casos foi necessário não seguir-se exatamente este diagrama: Considere o comando que define um tipo de objeto:

Sintaxe:

```
DEFINE OBJETO <tipo_objeto> ;
```

Este comando, definido desta forma, de acordo com a figura 6, permite apenas criar o objeto de nome "<tipo_objeto>" para o tipo "objeto". Apesar de ser obrigatório um relacionamento de modalidade "Habita", ao final deste comando esse relacionamento não é criado. Observe que tanto o tipo de objeto "colônia" quanto o tipo de objeto "objeto" tem

modalidades de relacionamento associando-os, e que são obrigatórias. Desta forma quando um destes objetos for criado deve-se criar também suas modalidades de relacionamento.

Resolveu-se este impasse estabelecendo-se que os tipos de objetos são definidos independentes de associa-los a algum tipo de colônia, de acordo com a sintaxe do comando acima, sendo que as associações deverão ocorrer através do comando que define um tipo de colônia.

Esta solução resolve o problema parcialmente, pois permite que objetos sejam definidos sem que se obrigue associá-los a qualquer colônia. Dessa forma, adotou-se a solução de utilizar a colônia Global onde os objetos inseridos por esse comando habitam.

Algumas regras podem ser tomadas como gerais para estabelecer a sintaxe de um comando:

- Todo objeto é identificado pelo seu tipo e seu nome, assim um comando da base que tenha como argumento um objeto, necessitará que seja fornecido no comando seu tipo e seu nome, através dos argumentos: "<tipo_objeto> <nome_objeto>". Exemplo:

```
LOCALIZA [OBJETO] <tipo_objeto> <nome_objeto> ;
```

No esquema o tipo de objeto é identificado de acordo com o elemento fornecido.

Exemplo:

```
DEFINE OBJETO <tipo_objeto> ;
```

Neste comando não foi fornecido pelo usuário o tipo de objeto, pois o tipo já faz parte do próprio comando "OBJETO". Assim o objeto a ser incluído é identificado pelo símbolo "OBJETO" e pelo nome "<tipo_objeto>".

- Todo relacionamento é identificado pela modalidade, objeto origem e objeto destino:

```
<modalidade> <tipo_objeto> <objeto_origem>
```

Assim o comando que manipula um relacionamento na base deve considerar estes argumentos.

Exemplo: o comando que localiza um relacionamento na base de dados tem a seguinte sintaxe:

```
LOCALIZA [RELACIONAMENTO] <modalidade>
```

<tipo_objeto_destino> <nome_objeto_destino> ;

Este comando tem como pré-condição que o objeto origem seja o corrente. Desta forma o objeto origem já foi fornecido previamente.

Já para o esquema, os comandos que necessitam manipular relacionamento o identificam de acordo com a operação solicitada pelo usuário, tais como: inclusão, eliminação, etc.; e pelos nomes dos objetos fornecidos no comando. Exemplo:

```
DEFINE COLONIA <tipo_colônia>  
CONSTRITA POR <tipo_objeto>  
HABITADA POR (<tipo_objeto>)* ;
```

Através deste comando um dos relacionamentos que pode ser identificado é:

Modalidade: "Constrita por";

Tipo/nome objeto origem: "colônia" "<tipo_colônia>";

Tipo/nome objeto destino: "objeto" "<tipo_objeto>".

4.7 - Conclusão.

Neste capítulo definiu-se a Linguagem de Acesso ao MRO, para ser aplicada ao GEO. Esta linguagem pode ser situada, de acordo com a classificação adotada no capítulo 2, entre as linguagens de base de dados orientadas a objetos definidas para um modelo.

Para definir a linguagem foram consideradas três abordagens: linguagens orientadas por menus, linguagens naturais e linguagem baseada em comandos e argumentos. Adotou-se a abordagem de linguagem baseada em comandos e argumentos, incorporando aspectos que aproximem aos da linguagem natural.

A garantia de que a linguagem representa fielmente o modelo de dados, reside no fato de que a construção sintática e semântica dos comandos foram baseadas no Meta-esquema do modelo, especificado através do DRO (figura do apêndice 2).

Capítulo 5

ASPECTOS DA IMPLEMENTAÇÃO DA LINGUAGEM

5.1 - Considerações gerais.

Neste capítulo é desenvolvido um estudo sobre a implementação da linguagem especificada no capítulo 4.

A implementação adotada tem características de um interpretador de comandos. Para a realização efetiva (análise semântica) dos comandos da linguagem, são utilizadas primitivas do Gerenciador de Objetos (GEO), desenvolvido no Instituto de Ciências Matemáticas de São Carlos - USP, o qual suporta parte dos conceitos definidos para o MRO.

A versão utilizada deste gerenciador foi a desenvolvida em microcomputador da linha PC-XT, tendo como sistema operacional MSDOS e a linguagem de programação "C".

A partir dos comandos propostos para a linguagem, selecionou-se um conjunto básico de comandos para serem implementados no GEO. Os comandos selecionados constituem aqueles que podem ser suportados pela versão utilizada do GEO, e possibilitam testar e mostrar a viabilidade da linguagem.

Além dos comandos propostos foram incluídos outros com a finalidade de oferecer facilidades para o usuário utilizar o interpretador.

A seguir é apresentado o subconjunto de comandos selecionados, e a extensão de comandos de auxílio ao usuário.

5.2 - Subconjunto de comandos implementado para o GEO.

A versão do gerenciador utilizada nesta implementação do interpretador não suporta a manipulação de:

- Relacionamento triplo;
- Supertipo;
- Subtipo;
- Gerenciamento de versões;
- Características de atributos: Gráfico, Tempo, Regra, Visualização, Estrutura e Procedimento. As características de atributos que podem ser manipuladas são Propriedade, Comentário e Sinônimo.

Os comandos que manipulam os conceitos listados anteriormente foram excluídos do subconjunto de comandos disponíveis nesta implementação. A tabela listada a seguir descreve de maneira geral o subconjunto de comandos presentes nesta implementação.

Categoria de Comandos	Comandos implementados
Inclusão	Todos com exceção dos que manipulam subtipo, supertipo, relacionamento triplo e característica de atributo comentário;
Eliminação	Todos com exceção dos que manipulam subtipo, supertipo, relacionamento triplo e característica de atributo comentário;
Substituição	Não foram implementados;
Estabelecimento de ambiente	Todos, exceto os que manipulam gerenciamento de versões.
Mostra	Todos comandos, exceto os que manipulam subtipo, supertipo, relacionamento triplo e característica de atributo comentário;
Seleção	Todos com exceção da manipulação de expressões envolvendo valores de atributos;
Lista / Máscara	Todos com exceção da definição de máscara junto com o comando "lista" e utilização de atributos intrínsecos na definição da máscara.

Tabela 1. Subconjunto de comandos implementados por categoria.

Nesta implementação não está presente a manipulação de objeto utilizando seu nome absoluto, isto é o interpretador não manipula argumentos que tenham nomes de objetos especificados usando seu caminho de acesso na hierarquia de colônias através do símbolo "/".

5.2.1 - Extensão dos comandos para auxílio ao usuário.

Os comandos listados a seguir não fazem parte da linguagem de acesso ao MRO propriamente dita, apenas foram incluídos na implementação deste interpretador para auxiliar o usuário na utilização de seus comandos.

As linguagens orientadas por comandos, como um todo, têm uma sintaxe bem definida, a qual o usuário deve memorizar. Desta forma, torna-se necessário oferecer algum comando que auxilie o usuário a lembrar-se da sintaxe do comando a ser utilizado. A sintaxe deste comando é definida da seguinte forma:

Sintaxe:

```
MANUAL ("<sequência_de_caracteres>") * ;
```

Este comando permite pesquisar a sintaxe dos comandos presentes nesta implementação. A sequência de argumentos fornecidos restringe a pesquisa.

Exemplo: Pesquisar a sintaxe do comando "DEFINE OBJETO".

```
MANUAL "define", "objeto";
```

ou

```
MANUAL "define";
```

Neste último caso, o comando exibirá a sintaxe de todos os comandos que iniciam-se com "DEFINE", isto é, todos os comandos que fazem a inclusão de um elemento no esquema.

Algumas operações executadas podem ser repetitivas, o que faz com que o usuário tenha que digitar o comando novamente. Desta forma torna-se adequado fornecer um comando que execute a operação de repetir comandos previamente digitados. Este comando é definido da seguinte maneira:

Sintaxe:

```
!(<número_do_comando>|!) ;
```

O usuário pode solicitar um comando já digitado fornecendo o número do comando, pois a cada comando é associado um número. Assim, após o comando de repetição "!", pode-se fornecer o número do comando ou digitar novamente "!", para indicar que o comando desejado é o último que foi digitado.

Para permitir ao usuário consultar os comandos já digitados com os seus respectivos números, definiu-se o comando no seguinte formato:

Sintaxe:

```
HISTORIA ;
```

A sintaxe de algum dos comandos anteriores de auxílio ao usuário é "inspirada" naqueles definidos para o interpretador de comandos "C-shell" do sistema operacional UNIX [Kernighan_84].

Para permitir uma maior versatilidade dos usuários para manipular este interpretador, foi definido um comando "LIGA" que altera alguns dos valores "defaults". Este comando apresenta as seguintes variações:

Sintaxe:

```
LIGA (ESQUEMA|BASE) ;
```

Através deste comando permite-se que o usuário da linguagem possa tornar corrente ora a colônia BEsquema (onde está definido o esquema do usuário) ora voltar para a colônia onde o usuário estava manipulando sua base de dados propriamente dita.

Outra variação do comando "LIGA" permite ao usuário alterar a apresentação do nome de objeto utilizada no comando "MOSTRA". A sintaxe deste comando é a seguinte:

Sintaxe:

```
LIGA "NOME" ("TIPO" | SO | RELATIVO | ABSOLUTO) ;
```

Observe que os argumentos "NOME" e "TIPO" estão entre aspas, pois são utilizadas como palavras reservadas em outros comandos.

O próximo comando foi acrescentado a linguagem com a finalidade de abandonar

as operações com o interpretador e voltar para o sistema operacional. Se houver alguma base de dados ativa ela será desativada. A sintaxe deste comando é a seguinte:

Sintaxe:

FINALIZA ;

5.3 - Aspectos da Gramática.

Para discutir a implementação dos comandos da linguagem, será necessário situar alguns conceitos manipulados pela implementação.

Os comandos da linguagem são especificados através de regras ou sentenças, que descrevem a estrutura de leitura (sintaxe) dos comandos. O conjunto destas regras compõe a gramática da linguagem. Após o reconhecimento de uma regra, é tomada uma ação correspondendo a realização do comando, a qual é denominada **ação semântica**. O reconhecimento de uma sentença da linguagem é feito pelo Analisador Sintático ou "parser", o qual faz o controle do fluxo de leitura dos itens ou símbolos fornecidos pelo usuário. A leitura dos comandos e a separação de cada item, denominado "token", é feita pelo Analisador Léxico. Os "tokens" reconhecidos podem ser classificados como: palavras reservadas, símbolos do usuário e separadores [Aho_77].

O **analisador sintático** foi desenvolvido utilizando o gerador de "parser" YACC [Kernighan_84], o qual gera uma rotina em linguagem "C" com o nome de "yyparser()", a partir de uma especificação da gramática;

O **analisador léxico** é uma rotina desenvolvida segundo as interfaces exigidas pelo YACC, denominada "yylex". Deve-se observar que os símbolos utilizados pelo YACC iniciam sempre por "yy";

O **analisador semântico** efetua a ação semântica, após o reconhecimento sintático das regras. Estas rotinas foram desenvolvidas utilizando as bibliotecas do GEO.

Segue-se a apresentação dos módulos que compõe a implementação da gramática.

5.3.1 - Analisador sintático.

Este analisador implementa a gramática especificada através de sua BNF [Nau_63],

Figura 7: Organização modular do interpretador.

listada no apêndice 3.

Os símbolos terminais da gramática desta linguagem estão representados pelos símbolos em letras maiúsculas, os quais estabelecem os símbolos que devem estar presentes no reconhecimento de uma regra. Fazem parte destes símbolos as palavras reservadas (apêndice 4), os separadores e operadores (apresentados no capítulo 4, no item 4.3.3). Os símbolos não terminais, quando presentes do lado direito de uma regra, denotam uma nova regra a ser reconhecida. Desta forma todos os símbolos não terminais estão presentes exatamente uma vez do lado esquerdo das regras.

Suponha que a seguinte regra represente o comando que define um tipo de objeto no esquema:

`<inc_de_t_de_obj> ::= DEFINE OBJETO <tipo_objeto> ;`

Um símbolo é delimitado através de separadores, representados no exemplo acima

pelo espaço em branco e pelo símbolo ";". A regra "<tipo_objeto>" é definida por outra regra denominada "nome", a qual define como um símbolo do usuário deve ser fornecido:

```

<tipo_objeto> ::= <cadeia>
<nome> ::= LETRA <números_letras>
          | <string>
<números_letras> ::= /* vazio */
                    | <número_letra>
                    | <números_letras> <número_letra>
<número_letra> ::= NUMERO
                 | LETRA
                 | _
<string> ::= "<caracteres>"
<caracteres> ::= CHARACTER
               | <caracteres> CHARACTER

```

Desta forma, um nome fornecido pelo usuário como argumento em algum comando, deve iniciar por uma letra seguida ou não de letras, ou números ou do símbolo "_". A única restrição é que esta sequência de caracteres não pertença ao conjunto de palavras reservadas da gramática. Uma forma alternativa de definir um nome é através da regra "<string>", a qual define uma sequência de quaisquer caracteres entre aspas. Isto permite que o usuário possa fornecer como argumento do comando um nome que é uma palavra reservada ou mesmo uma sequência de caracteres que contenha separadores. Esta solução foi adotada para fornecer uma maior flexibilidade ao usuário, no que se refere ao uso de palavras reservadas fora de seu contexto.

Exemplo: através desta gramática pode-se fornecer o seguinte comando:

```
DEFINE OBJETO "considera";
```

As regras que mapeiam os comandos da linguagem estão agrupadas em quatro categorias: esquema, base de dados, estabelecimento de ambiente e de extensão para auxílio ao usuário. Desta forma um comando da linguagem pode ser definido através da seguinte regra:

```

<comando> ::= <comando_esquema>
              | <comando_base>
              | <comando_ambiente>
              | <comando_extensão>

```

A partir da primeira regra da gramática são derivadas todas as demais regras. Esta regra define o Símbolo Inicial da Linguagem.

```
<lamro> ::= /* vazio */
           | FINALIZA ;
           | <lamro>
           | <comandos>
<comandos> ::= <comando>
              | <comandos> <comando>
```

O analisador sintático desta linguagem foi implementado utilizando a ferramenta de programação YACC, através do mapeamento da especificação da gramática listada no apêndice 3, para a especificação exigida pelo YACC. Essa especificação está listada no apêndice 5. Uma revisão mais completa sobre a utilização do YACC na implementação da gramáticas pode ser encontrada em [Schreiner_85] e [Kernighan_84].

O analisador sintático é gerado a partir da especificação da gramática fornecida ao YACC. A rotina gerada em linguagem "C" chama-se "yyparser()". Toda vez que essa rotina necessita de um símbolo terminal, ela chama a rotina "yylex()", a qual devolve o código e os caracteres que compõe o símbolo.

Segue-se a especificação em YACC para a regra "<inc_de_t_de_obj>".

```
%token DEFINE OBJETO
%% /* delimita o início da especificação da linguagem */
inc_de_t_de_obj: DEFINE OBJETO ts_objeto ';' ;
ts_objetos: t_objeto
            | ts_objetos t_objeto
            ;
t_objeto: NOME
          | STRING
          ;
%% /* delimita o final da especificação da linguagem */
```

Os símbolos terminais que são identificados pelo analisador léxico, devem ser obrigatoriamente declarados através da cláusula "%token". O símbolo terminal que tem somente um caracter já determinado não necessita ser declarado, e deverá estar presente na regra entre

aspas simples (';'). A regra que define um nome e uma sequência de caracteres ("string"), não está presente na especificação da gramática para o YACC. Isto ocorre porque essa análise foi transferida para o analisador léxico.

5.3.1.1 - Prioridade e associatividade dos operadores.

Foram estabelecidas para a linguagem quatro categorias de operadores, para as quais foram estabelecidas as prioridades e associatividades, especificadas através do YACC. A prioridade entre os operadores é especificada pela sequência vertical em que aparecem, sendo que aqueles que estão na mesma linha têm a mesma prioridade. Já a associatividade é indicada por "left", significando da esquerda para direita, e "right" da direita para esquerda. A seguir será listada a prioridade e a associatividade atribuída aos operadores da gramática, utilizados pela sintaxe que manipula seleção:

```
%left QUE_NAO_TEM_ATRIBUTO QUE_TEM_ATRIBUTO /* ~-$ -$ */
%left TIPO
%left DESTINO
%left RELACIONAMENTO
                                     /* ~-$ -$ */
%left QUE_NAO_TEM_RELACIONAMENTO QUE_TEM_RELACIONAMENTO
%left '^^'
%left UNIAO DIFERENCA /* + - */
```

Alguns dos operadores descritos anteriormente são conhecidos pelo usuário através de seu símbolo, tais como "UNIAO" em que é utilizado o símbolo "+", "QUE_TEM_RELACIONAMENTO" e "QUE_TEM_ATRIBUTO" correspondem ao operadores "~-\$" e "-\$" respectivamente.

Exemplo: Considere a seguinte sequência de comandos:

```
DEFINE SELECAO;
INCLUI DESTINO TIPO Equipamento -$ ATRIBUTO Valor
RELACIONAMENTO Utilizado TIPO Projeto;
```

A sequência em que os comandos são executados é a seguinte:

1) DEFINE SELECAO (cria uma seleção temporária $\alpha 0$);

- 2) TIPO Equipamento (obtem a seleção temporária α_1);
- 3) α_1 -\$ ATRIBUTO Valor (são removidos de α_1 os objetos que não têm o atributo valor);
- 4) TIPO Projeto (obtem a seleção temporária α_2);
- 5) DESTINO α_1 RELACIONAMENTO Utilizado α_2 (resulta na seleção α_2 e elimina a seleção α_1);
- 6) INCLUI α_2 (copia a seleção temporária α_2 na seleção α_0).

Deve-se observar que para obter a prioridade e associatividade dos operadores pode-se utilizar de dois artifícios [Shereiner_85]: através da definição da gramática; ou quando a gramática não puder especificá-las, utilizando-se da declaração externa a gramática, de acordo com a notação exibida anteriormente.

A prioridade de reconhecimento de operadores pertencentes à mesma categoria pode ser alterada utilizando parenteses.

As especificações da gramática para o YACC apresentadas neste item não tem a intenção de abranger todos os recursos utilizados do YACC, mas apenas mostrar alguns aspectos envolvidos no mapeamento da especificação da gramática da linguagem para a sua especificação utilizando a ferramenta de programação YACC.

5.3.2 - Analisador Léxico.

Toda vez que é ativado o analisador léxico, denominado "yylex()", este retorna na estrutura de dados global "yylval", definida pelo YACC, o valor do símbolo identificado e seu código que também é definido pelo YACC através de uma "macro" ("define") da linguagem "C".

Exemplo: O trecho de código a seguir ilustra a leitura de um separador.

```
#include <mro.h>
% token LE
%% /* inicio da especificação da gramática */
.
.
%% /* final da especificação da gramática */
.
.
yylex()
```

```

{
    int c, codigo_simbolo;
    cadeia p; /* auxilia a formar a sequencia de
               caracteres do separador */
               */

    .
    .
    if ((c = getchar()) == '+') {
        codigo_simbolo = UNIAO;
        p = (unsigned char *) emalloc(3);
        p[0] = p[1] = 1;
        p[2] = '+';
    }
    yylval.simbolo = p; /* valor do separador */
    return codigo_simbolo; /* codigo do separador */
}

```

Para fazer a manipulação interna dos símbolos utilizou-se um tipo de dado definido na biblioteca do GEO que faz a manipulação de sequência de caracteres denominada de Cadeia. Esta estrutura pode ser entendida como um vetor, com algumas características especiais. Considere o exemplo acima da cadeia "p":

p[0] --> quantidade máxima de caracteres;
 p[1] --> quantidade de caracteres presentes.

Assim quando a cadeia "p" tem o valor de "oi", cada um dos índices do vetor poderia ter os seguintes valores: p[0] == 2, p[1] == 2, p[2] == 'o' e p[3] == 'i'.

De maneira geral, o analisador léxico reconhece:

- Todo conjunto de separadores, utilizados na definição da linguagem;
- Um caracter delimitado por aspas simples. Exemplo: 'a', '\'', '\', etc;
- Número inteiro ou no formato de ponto flutuante;
- Uma palavra reservada da gramática;
- Um símbolo do usuário, que define uma sequência de caracteres que podem estar delimitados por aspas duplas ("string") ou não.

O analisador léxico pode gerar alguns erros, os quais ocorrem quando o limite de tamanho máximo permitido para algum símbolo é excedido, como por exemplo para um número.

5.3.3 - Analisador semântico.

As ações semânticas são ativadas pelo "parser" após o reconhecimento de alguma regra. Desta forma, após o reconhecimento de um comando fornecido pelo usuário, a rotina correspondente à execução da ação semântica desse comando é ativada com os argumentos presentes no comando. Esta rotina deve ser fornecida junto com a especificação da gramática para o YACC. A seguir será mostrado um exemplo do código utilizado pelo YACC para um comando que realiza a inclusão de um objeto na base de dados:

```
inc_de_obj: INCLUI OBJETO t_objeto objetos ';'
{
    int err, i;
    err = acao100($2, lst, qtd_lst);
    if (err != 0){
        erro("erro semantico. ", err, lineno);
    }else oque = 0;
    (void) free($2); /* realocacao de tipo */
    for (i = 0; i <= qtd_lst; i++){
        (void) free(lst[i]);
    }
}; /* Fim da acao */
```

Após o reconhecimento da regra que define o comando, a "acao100" é executada com os seguintes parâmetros:

- "\$2": representa uma sequência de caracteres (tipo cadeia) obtida através do reconhecimento da regra "nome";
- "lst": vetor de cadeias que armazena todos os nomes de objetos fornecidos durante o reconhecimento da regra "objetos";
- "qtd_lst": inteiro que armazena a quantidade de cadeias presentes no vetor "lst".

A rotina "ação100" é responsável pela ação semântica deste comando, fazendo a interação com as rotinas de biblioteca do GEO, para inclusão de objetos de um tipo. A listagem a seguir mostra a implementação desta rotina:

```
#include <mro.h>
int acao100(tipo, arCadeias, qtd_lst)
```

```

cadeia tipo,      /* Tipo dos objetos a serem incluídos na base */
arCadeias[];      /* Array de cadeia de objetos */
int qtd_lst;
{
    int i;          /* índice do array de cadeia de objetos */
    int  codigo_corrente; /* recebe o status de uma operação */
    coditipo  cto;     /* indica o tipo do objeto */
    /* Descobre o código do tipo de objeto */
    if( (codigo_corrente = CodOBJ(tipo, &cto)) && \
        (codigo_corrente != 0)) return codigo_corrente;
    /* recebe todos objetos a serem incluídos */
    for (i = 0; i <= qtd_lst; i++){
        if((codigo_corrente = CTOBJ(arCadeias[i],cto)) \
            && (codigo_corrente != 0)) return codigo_corrente;
    }
    return 0;
}

```

Nesta ação são utilizadas duas rotinas da biblioteca do GEO, mostradas a seguir:

- "CodOBJ": dado o nome do tipo de objeto, retorna o código do tipo;
- "CTOBJ": inclui um objeto na base de dados para o código do tipo fornecido.

O objeto será incluído na colônia corrente e o último objeto incluído permanecerá como o objeto corrente.

As características das ações semânticas relacionadas com a atuação dos comandos, descritas no capítulo 4, são fornecidas pelas rotinas do GEO. Assim por exemplo, a característica de tornar corrente o último objeto incluído na base de dados, é fornecida pela rotina "CTOBJ", que inclui um objeto e o torna corrente.

As ações semânticas de maneira geral não emitem mensagens de erro: quando necessário apenas devolvem um código de erro. As informações sobre o erro serão exibidas ao usuário sob o critério da rotina que a ativou, como pode ser observado no exemplo anterior. A "acao100" não emite nenhuma mensagem de erro, apenas informa o código do resultado da operação. Já o bloco de instruções correspondentes as ações semânticas ativadas após o reconhecimento do comando, avalia se houve algum erro e exibe as informações ao usuário, ativando a rotina "erro".

De uma forma geral as rotinas "açãoXXX" foram agrupadas em módulos distintos,

assim sendo, as rotinas que manipulam objeto estão definidas no módulo "AO.C", as rotinas que manipulam relacionamento estão definidas no módulo "AR.C", etc.

As rotinas que efetuam as ações semânticas dos comandos, estão agrupadas nos seguintes módulos:

"AO.C":	rotinas que fazem a manipulação ¹ de objeto;
"AR.C":	rotinas que fazem a manipulação de relacionamento;
"AA.C":	rotinas que fazem a manipulação de atributo;
"AC.C":	rotinas que fazem a manipulação de colônia;
"AM.C":	rotinas que implementam os comandos "mostra";
"AS.C":	rotinas que implementam os comandos de "seleção";
"AMA.C":	rotinas que implementam os comandos de "máscara/lista";
"AG.C":	outros módulos implementados como ações semânticas auxiliares, como por exemplo as rotinas de ambiente.

5.3.3.1 - Operações sobre atributos.

Algumas operações que manipulam atributo na base de dados necessitam identificar sobre que elemento (objeto ou relacionamento) a operação refere-se. Por exemplo, algumas ações do comando "MOSTRA", que exibem algum valor de atributo ou mesmo as operações dos comandos que incluem ou removem algum atributo da base de dados, necessitam identificar sobre que elemento deverão atuar.

Para o usuário ter certeza que a operação será realizada sobre o elemento desejado, alguns comandos da linguagem oferecem uma cláusula onde o usuário indica (de forma optativa) o elemento sobre qual a operação irá atuar. Por exemplo:

```
INCLUI ATRIBUTO RELACIONAMENTO Data_inicio "10/10/90";
```

O interpretador mantém a informação de qual elemento foi manipulado por último. As manipulações que afetam o elemento corrente (objeto ou relacionamento) são as de inclusão. Assim, após a localização de um relacionamento, o elemento sobre o qual ocorrerão as operações

---

¹ Por manipulação entende-se inclusão, eliminação e substituição.

de atributo será por "default" o relacionamento. Também quando o elemento é indicado explicitamente em um comando, como no exemplo anterior, este permanecerá ativo para o próximo comando.

Quando as operações são iniciadas em uma base de dados, o elemento ativo é objeto.

5.4 - Características da gramática para utilização do YACC.

Através da utilização do YACC possibilitou-se o refinamento da gramática de forma a garantir o funcionamento correto do analisador sintático gerado pela ferramenta. As características e os conceitos levantados neste item podem ser encontrados em [Shreiner_85].

Para todo símbolo não terminal presente na gramática deve haver uma regra que o defina, sendo que todo símbolo não terminal precisa ser derivado a partir de um único símbolo não terminal, denominado símbolo inicial da gramática.

A gramática não deve ser ambígua, isto é, para uma sequência de símbolos terminais, só deve haver uma sequência de derivações de regras (símbolos não terminais) que os reconhecem. Caso contrário a sequência de comandos não pertence a gramática. Utilizando o YACC para análise da linguagem as regras ambíguas podem ser detectadas.

Além de resolver as ambiguidades, outra característica importante da gramática é ser determinística. Durante o reconhecimento de uma sequência de símbolos terminais, o analisador vai montando a sequência de regras que reconhecem estes símbolos. Esta sequência é denominada árvore de parser. Durante o processo de montagem, não deve ocorrer "backtracking", isto é, não deve ocorrer que o caminho seguido na árvore esteja errado, voltando algumas regras anteriores para tentar percorrer um outro ramo de caminho de regras. Utilizando o YACC, regras que tenham este comportamento podem ser identificadas.

Outra característica fundamental é que a gramática deve ser do tipo LL(1). Este tipo de gramática define que a árvore de "parser" será construída a partir do símbolo inicial, isto é, será construída através do processo "top-down", sem utilizar "backtracking". A decisão do caminho a seguir na construção da árvore de "parser" deve ser através da leitura de apenas um símbolo. Em outras palavras, indicando que sempre que haja uma meta (símbolo terminal ou não) que uma regra utilize, o próximo símbolo lido deve habilitar o analisador a resolver o que

fazer.

5.5 - Considerações sobre a utilização do YACC.

Através da utilização da ferramenta YACC para gerar o analisador sintático, foram observadas as seguintes vantagens:

- Possibilidade de depurar a gramática, por exemplo, detectando suas ambiguidades. Para resolvê-las em alguns casos necessitou-se fazer alterações na sintaxe de algum comando;
- Facilidade de efetuar alterações na gramática da linguagem. As alterações ocorridas foram facilitadas já que ocorrem a nível da própria especificação da gramática;
- Facilita a modularização. A ferramenta oferece facilidades para que haja uma separação da análise sintática, da léxica e da semântica, tornando o código mais claro;
- Torna mais produtivo o trabalho de implementação do interpretador, visto que facilita a construção do analisador sintático, e o código gerado pelo YACC é confiável.

No caso de interpretadores de comandos simples, onde as regras são compostas em sua maioria por símbolos terminais, o YACC pode gerar um código maior do que se fosse desenvolvido o código sem auxílio do YACC. Quando o principal requisito do interpretador for eficiência, essa questão poderia ser tratada desenvolvendo-se um analisador sintático para as regras mais simples e para as mais complexas (por exemplo regras que manipulam seleção), utilizar o YACC para gerar o código do "parser". Deve-se lembrar que esta solução implica em outros problemas, tais como, estruturas de dados que executem funções semelhantes nos dois analisadores, mas constitui numa solução de fácil implementação.

Esta questão foi constatada a partir da observação de códigos de dois interpretadores, os quais implementavam o mesmo conjunto de comandos com regras gramaticais simples. O primeiro denominado "GEO_INC", desenvolvido sem a utilização da ferramenta. O segundo desenvolvido com o uso do YACC, utilizando as mesmas ações semânticas e analisador sintático semelhante.

5.6 - Conclusão.

O interpretador de comandos gerado executa a análise sintática de um subconjunto de comandos definidos no capítulo 4. No entanto somente os comandos cuja a ação semântica é suportada pela versão utilizada do GEO foram implementados.

Apesar de procurar-se adotar um padrão e símbolos significativos na definição dos comandos, devido à grande quantidade de operações e conceitos manipulados pela linguagem, resultou na definição de um grande número de comandos, criando dificuldades para o usuário memorizar sua sintaxe. Desta forma tornou-se necessário incluir comandos de auxílio na linguagem para a sua utilização. Os comandos incluídos foram inspirados nos comandos do "C-shell" do UNIX, com algumas variações. Além da extensão de comandos para auxílio do usuário, outros poderiam ser incluídos, principalmente aqueles que permitam editar alguma linha de comando já fornecido.

A notação utilizada para a especificação da gramática facilitou a sua transcrição para a notação do YACC. Foram adotados critérios para agrupar as regras, as quais buscam facilitar a inclusão de regras para novos comandos que por ventura venham a ser definidos.

A prioridade e associatividade dos operadores foram obtidas através de construções gramaticais, e quando não pode-se especificá-las através da gramática, utilizavam-se declarações do YACC externas a gramática.

Observou-se que a utilização de ferramentas de programação do ambiente UNIX permitiram alcançar uma melhor eficiência nas atividades de codificação. O uso do YACC facilitou a implementação do analisador sintático, bem como facilita a evolução e manutenção da gramática da linguagem, na medida em que o YACC gera o analisador a partir de uma especificação da gramática. Por outro lado, para regras que definem a sintaxe de comandos simples, o YACC gera um código ineficiente. A solução para esta questão seria construir um misto entre um analisador para as regras mais simples desenvolvido sem a utilização da ferramenta e usa-la nas regras mais complexas, para gerar o analisador sintático.

Capítulo 6

DECISÕES DE PROJETO, SUGESTÕES DE FUTURAS PESQUISAS E CONCLUSÕES

6.1 - Considerações gerais.

Este capítulo destaca a síntese dos estudos realizados neste trabalho. Ressalta algumas decisões de projeto, sugestões de futuras pesquisas que podem vir a ser desenvolvidas a partir deste trabalho e as conclusões gerais.

6.2 - Decisões de projeto.

Neste item são feitas as descrições de algumas das decisões de projeto assumidas durante este trabalho, de forma a ressaltar aquelas já descritas no texto e complementar outras que não foram anteriormente citadas.

Durante a definição da linguagem procurou-se abranger todos os conceitos do MRO, os quais foram mapeados em comandos de uma maneira uniforme.

Os vários requisitos da linguagem, tais como: o de simplicidade e facilidade de utilização, disponibilidade do interpretador no ambiente operacional do GEO, entre outros, levaram a optar por uma linguagem de base de dados orientada por comandos. Sendo esta a primeira linguagem definida para o GEO, a abordagem adotada tornou-se adequada pois possibilitou cobrir os conceitos do MRO, devido a características da linguagem de comandos de permitir que o argumento fornecido no comando indique qual o elemento do modelo a operação deverá atuar. Por exemplo: o argumento fornecido após o comando "INCLUI" define o elemento do modelo que recairá a operação de inclusão, e também os argumentos restantes necessários para completar

a especificação do comando.

Dentre o conjunto de comandos selecionados para serem implementados, optou-se por excluir os comandos que manipulam relacionamento triplo. Tomou-se esta decisão pois a definição de relacionamento triplo não encontra-se consolidada.

Apesar dos comandos MOSTRA e LISTA apresentarem algumas semelhanças de funcionalidade, deve-se considerar uma distinção da natureza destes comandos. O comando MOSTRA está mais na direção de "Browsers" no sentido de permitir situar o usuário da linguagem segundo os diferentes elementos do modelo presentes na base. O comando LISTA é poderoso na direção de permitir a construção de geradores de relatórios.

Com respeito a sintaxe dos comandos optou-se por flexionar os verbos no tempo presente do indicativo, na terceira pessoa do singular, por exemplo: MOSTRA, LISTA, ELIMINA, etc. Poder-se-ia adotar outras soluções, utilizando os verbos no infinitivo, tais como MOSTRAR, LISTAR, ELIMINAR. Outra solução seria utilizar o imperativo, tais como: MOSTRE, LISTE, ELIMINE, etc.

Com o objetivo de proporcionar uma maior legibilidade em alguns comandos, adotou-se algumas soluções:

- Reconhecimento de palavras reservadas no plural, por exemplo: "RELACIONAMENTOS", "ACESSIVEIS", etc;
- Utilização opcional de palavras reservadas que representam conectivos, tais como: "POR", "NA", "DE", etc.

Como na linguagem o conjunto de palavras reservadas é grande, tentou-se minimizar o problema da restrição da utilização delas como símbolos do usuário, através do uso de aspas para o símbolo, sendo caracterizado assim como "string". Outra solução que poderia ser adotada, seria de condicionar a análise léxica a algum contexto sintático. Por exemplo, nesta situação poderia-se incluir um novo tipo de objeto com o nome *define* (palavra reservada); assim o símbolo "define" apareceria duas vezes num mesmo comando sendo a primeira vez como nome de comando e a segunda como símbolo do usuário. Embora esta solução possa ser implementada utilizando o YACC, segundo [Johnson_78] deve-se evita-la, pois pode tornar a especificação do analisador sintático complexa, perdendo uma das principais vantagens de utilização do YACC, que é a clareza de sua especificação.

Adotou-se a solução de não reconhecer abreviações dos comandos, por exemplo, "RELACIONAMENTO" ser reconhecido por "RELACIO", esta solução é utilizada em

interpretadores com o objetivo de trazer mais simplicidade para o usuário na digitação dos comandos. Os interpretadores que reconhecem abreviações mínimas, utilizam uma técnica de análise léxica/sintática diferente da empregada neste trabalho, e adotar esta solução implicaria em aumentar ainda mais o inconveniente do reconhecimento de palavras reservadas em situações não desejadas.

A decisão pela utilização do gerador de analisador sintático YACC, foi devido principalmente a clareza de sua especificação, pois desta forma facilita a evolução da linguagem.

6.3 - Sugestões de futuras pesquisas.

Neste item apresentam-se sugestões de outros estudos que podem ser motivados a partir desta pesquisa.

Uma extensão deste trabalho é a utilização de comandos de seleção mesclados com outros comandos. Por exemplo, o comando de inclusão de relacionamento poderia ser definido da seguinte forma:

Sintaxe:

```
INCLUI [RELACIONAMENTO] <modalidade> <seleção> ;
```

Desta forma outros comandos da linguagem, poderão obter também a capacidade de expressividade dos comandos de seleção.

Utilizando a definição da linguagem e parte do interpretador pode-se continuar este trabalho com o objetivo de implementar um pré-processador desta linguagem, de forma a possibilitar a utilização da LAMRO embutida em outras linguagens de programação, tais como, C ou C++.

Outra possibilidade é a utilização do analisador para gerar trechos de código, principalmente a definição de estruturas de dados, que possam ser incorporadas em conjuntos de aplicativos possibilitando o desenvolvimento de aplicativos baseados em domínios de aplicações.

Uma outra abordagem que pode ser seguida é a de incluir na linguagem instruções que permitam o controle de fluxo, ou seja, instruções que não estão diretamente relacionadas com a manipulação da base de dados, mas que são importantes no desenvolvimento de programas. Nesta mesma direção pode-se incluir instruções que permitam a definição de "macros" de

comandos da linguagem.

É possível ainda, estudar formas de representar na linguagem a manipulação de atributos gráficos

Devido a grande difusão da linguagem SQL em SGBDs convencionais, alguns SGBDs que utilizam conceitos de objetos têm adotado extensões SQL como linguagem de consulta, como pode ser observado no estudo realizado no capítulo 2. Assim torna-se importante implementar também uma linguagem de consulta baseada na sintaxe de SQL. Esse trabalho permitiria comparar com mais precisão os comandos de seleção propostos neste trabalho com a linguagem SQL.

Além disso, pode-se citar como sugestões:

A definição de comandos que permitam a manipulação de características gráficas de atributos e

A utilização de janelas e "clipboards" na execução dos comandos da linguagem, principalmente nos comandos MOSTRA e LISTA.

6.4 - Conclusões.

As pesquisas em linguagens de base de dados vêm acompanhando a evolução dos modelos e sistemas gerenciadores de base de dados, na medida que permitem manipular os conceitos definidos no modelo de dados e suportados pelo gerenciador (SGBD).

A Linguagem de Acesso ao MRO proposta neste trabalho, segue a abordagem orientada a comandos em detrimento das abordagens orientada por menus e natural.

Esta linguagem permite facilitar tanto a definição do esquema de uma aplicação, como também, as manipulações na base de dados propriamente dita. A utilização do GEO torna-se mais natural evitando, desta forma, que os usuários do gerenciador necessitem criar aplicações em linguagem C, através das rotinas da biblioteca do GEO, para fazer manipulações numa base de dados. Esta linguagem também facilita um melhor entendimento dos conceitos que envolvem o Modelo de Representação de Objetos, pois estes conceitos estão expressos na sintaxe e na semântica dos comandos.

Os comandos de seleção de objetos, presentes nesta linguagem, permitem construir consultas na base de dados segundo os diferentes aspectos do modelo, tais como,

relacionamentos, atributos, colônias, tipo de objeto, etc. Isto permite que estes comandos façam uso da própria semântica da informação, manipulada na base de dados, para formular consultas que utilizem de vários recursos presentes no MRO, contrapondo-se com a linguagem SQL, onde os conceitos restringem-se basicamente a relações e campos.

Os comandos de máscara constituem recursos ao usuário que permitem agrupar as informações armazenadas na base de dados segundo formatos diferentes daqueles que estão armazenados na base de dados. Observados desta forma, estes comandos têm características que permitem identifica-los como sendo os de um Gerador de Relatórios.

A implementação da linguagem, descrita no capítulo 5, serve como subsídio para mostrar uma experimentação e também a viabilidade da proposta apresentada. Desta forma, esta implementação foi desenvolvida como um interpretador de comandos, utilizando microcomputador da linha PC-XT, baseado no Sistema Operacional MS-DOS, de acordo com o ambiente operacional utilizado pela versão em uso do GEO. Foi utilizada também a linguagem de programação "C" e outras ferramentas do ambiente de programação UNIX, dentre as quais pode-se destacar o gerador de analisadores sintáticos YACC.

REFERÊNCIAS BIBLIOGRÁFICAS

- [AHO_77] AHO, A.V. e ULLMAN, J.D. *Principles of the compiler design*, Mass: Addison-Wesley, 1977.
- [AKHRAS_89] AKHRAS, F.A. e COSTA, M.C. *Meta-modelo para ferramentas genéricas integráveis ao ambiente SIPS*. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 3, Recife, out 1989. p203-215. *Anais*.
- [ANNEVELINK_91] ANNEVELINK, J. Database programming languages: a functional approach. *Communications of the ACM*, v.15, n.11, p318-327, nov 1991.
- [ASTRAHAN_75] ASTRAHAN, M.M. e CHAMBERLIN, D.D. Implementation of a structured english query language. *Communications of the ACM*, v.18, n.10, p580-588, out 1975.
- [BARBEDETTE_86] BARBEDETTE, G. e RICHARD, P. *VOOD: the object-oriented data model*. Institut National de Recherche en Informatique et Automatique, Paris, p27, 1986.
- [BATORY_88] BATORY, D.S. et alli Implementation concepts for an extensible data model and data language. *ACM Trans Database Systems*, v.13, n.8, p231-262, set 1986.
- [BOYCE_73] BOYCE, R.F. et alli. Specifying queries as relational expressions: SQUARE. *ACM SIGPLAN/SIGIR*, v.10, n.3, p103-135, nov 1973.
- [CAREY_88] CAREY, M.J. et alli. A data model and query language for EXODUS. *Communications of the ACM*, v.13, n.8, p413-423, set 1988.
- [CATTELL_91] CATTELL, R.G.G. What are next-generation database system?. *Communications of the ACM*, v.34, n.10, p31-33, out 1991.

- [CALÓNEGO_91] CALÓNEGO Jr, N. *Desenvolvimento de um núcleo multi-usuário para um sistema de gerenciamento de base de dados orientada a objetos*. São Carlos: USP/ICMSC, 1991. 133p. (Dissertação de Mestrado).
- [CHAMBERLIN_74] CHAMBERLIN, D.D. e BOYCE R.F. SEQUEL: a structured query language. ACM SIGMOD WORKSHOP ON DATA DESCRIPTION, ACCESS AND CONTROL, n.10, v.5, 1974. *Proceedings*.
- [CHAMBERLIN_76] CHAMBERLIN, D.D. et alli. SEQUEL2: a unified approach to data definition, manipulation, and control. *IBM J. Res. Develop*, v.20, n.6, p560-575, nov 76.
- [CHEN_76] CHEN, P.P. The entity-relationship model - toward a unified view of data. *ACM Trans Database Systems*, n.13, v.6, p9-36, mar 1976.
- [CODD_70] CODD, E.F. A relational model of data for large shared data banks. *Communications of the ACM*, v.13, n.6, p377-387, jun 1970.
- [CODD_79] CODD, E.F. Extending the database relational model to capture more meaning. *ACM Trans on Database Systems*, v.4, n.4, p33-44, dez 1979.
- [COFFIN_88] COFFIN, S. *UNIX the complete reference*. Bekerley: McGraw-Hill, 1988. p704.
- [COPELAND_84] COPELAND, G. e MAIER, D. Making a smalltalk a database system, *Communications of the ACM*, v.5, n.8, p316-325, 1984.
- [DATE_86] DATE, C.J. *An introduction to database systems*. Trad. Hélio Auro Gouveia. 4.ed Rio de Janeiro: Campus, 1986. p513.
- [EARL_85] EARL, A.N. e WHITTINGTON, R.P. Capturing the semantics of an IPSE. *Database Systems*, v.27, n.9, p33-43, nov 1985.
- [FERREIRA_91] FERREIRA, J.E. *Estudo da distribuição de uma base de dados apoiada no*

- Modelo de Representação de Objetos*. São Carlos: USP/IFQSC, 1991. 113p. (Dissertação de Mestrado).
- [GOLDBERG_83] GOLDBERG, A. e ROBSON, D. *Smalltalk-80: the language and its implementation*. Mass: Addison-Wesley, 1983.
- [HEERJEE_88] HEERJEE, B.K. e SADEGHI R. Rapid implementation of SQL: a case study using YACC and LEX. *Information and Software Technology*, v.30, n.3, p288-236, 1988.
- [JACKSON_91] JACKSON, M.S. Tutorial on object-oriented databases. *Information and Software Technology*, v.33, n.1, p4-12, 1991.
- [JOHNSON_78] JOHNSON, S.C. *YACC: yet another compiler-compiler*. New Jersey: Bell Laboratories, 1978, p24.
- [KERNIGHAN_84] KERNIGHAN, B.W. *The UNIX programming environment*. New Jersey, Prentice-Hall, 1984, p357.
- [KORTH_89] KORTH, H.F. e SILBERSCHATZ, A. *Sistemas de bancos de dados*. São Paulo: McGraw-Hill, 1989, p582.
- [LANA_90] LANA NETO, José Dias. *Signo-c: Uma ferramenta para Integração de uma fábrica de software*. São José dos Campos: ITA, 1990. 186p. (Dissertação de Mestrado).
- [MALHOTRA_89] MALHOTRA, A. et alli An entity-relationship programming language. *IEEE Trans on Software Engineering*, v.15, n.9, p1120-1130, set 1989.
- [MAMEDE_88] MAMEDE, C.A.H. e FALCÃO, S.D. *Linguagem de recuperação de informação*. Brasília, 1988. 77p. (Disciplina Estágio supervisionado do curso de Processamento de Dados. Instituto de Ciências Exatas - UNB).
- [MEIER_83] MEIER, D. *The theory of relational databases*. Rockville: Computer Science Press,

1983. p673.

[MELO_88] MELO, R.N. *Banco de dados não convencionais: a tecnologia do BD e suas novas áreas de aplicação*. VI Escola de Computação, Campinas, 1988.

[NAUR_63] NAUR, P. Revised report on the algorithmic language algol60, *Communications of the ACM*, v.2, n.1, p1-17, jan 1963.

[PRICE_89] PRICE, R.T. e GOLENDZINER, L.G. *Banco de dados para aplicações não convencionais*. Buenos Aires: Kapelusz, 1989. p165.

[RICHARDSON_89] RICHARDSON, J.E. e CAREY, J. Persistence in the E language issues and implementation. *Software Practice and Experience*, v.19, n.12, p1115-1150, 1989.

[ROBERTS_91] ROBERTS, S.A. et alli Application of object-oriented databases to geographic information systems. *Information and Software Technology*, v.33, n.1, p38-46, 1991.

[RODRIGUEZ_90] RODRIGUEZ, M. e MASIERO, P.C. *Apoio por computador à criação de especificações operacionais*. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 4, Águas de São Pedro, out 1992. p203-216. *Anais*.

[ROWE_87] ROWE, L. e STONEBRAKER, M. *The Postgres data model*. In: CONF. VERY LARGE DATA BASES, Brighton, 1987. *Proceedings*. p83-96.

[SALVIANO_92] SALVIANO, C.F. et alli *Signo: the SPP object management system*. In: ANNUAL ENERGY - SOURCES TECHNOLOGY CONFERENCE & EXIBITION, 15, Houston, jan 1992. *Proccedings*.

[SCHREINER_85] SCHREINER, A.T. e FRIEDMAN Jr, H.G. *Introduction to compiler construction with UNIX*. New Jersey: Prentice-Hall, 1985. p187.

[SMITH_87] SMITH, K.E. e ZDONIK, S.B. Intermedia: a case study of difference between

relational and object oriented. *Database Systems - ACM Proceedings*, v.4, n.8, p452-465. out 1987.

[SOBRINHO_87] SOBRINHO, F.G. e FERRARETTO, M.D. *Software plant: the Brazilian Software Consortium*. FALL JOINT COMPUTER CONFERENCE, IEEE, Dallas, 1987. *Proceedings*.

[STONEBRAKER_76] STONEBRAKER, M. et alli. The design and implementation of INGRES. *ACM Trans Database Systems*, v.1, n.3, p189-222, set 1976.

[STONEBRAKER_86] STONEBRAKER, M. et alli. *The design of Postgres*, In: PROC 1986 ACM-SIGMOD INT CONF MANAGEMENT OF DATA, jun 1986. *Proceedings*. p340-355.

[STROUSTRUP_86] STROUSTRUP, B. *The C++ programming language*, Reading: Addison-Wesley, 1986.

[SUTHERLAND_89] SUTHERLAND, J.V. *Is a persistent object store a database ?*. OOPSLA'89, out 1989. *Proceedings*, p499-503.

[TAKAI_91] TAKAI, O.K. *Representação de atributos com Características gráficas em uma base de dados orientada a objetos*. São Carlos: USP/ICMSC, 1991. p46. (Mini-dissertação apresentada para o exame de qualificação de Mestrado).

[TRAINA_82] TRAINA Jr., Caetano. *Sistema de Apoio por computador à documentação de sistemas*. São Carlos: USP/ICMSC, 1982. p379. (Dissertação de Mestrado).

[TRAINA_86] TRAINA Jr, Caetano. *Modelo de base de dados e máquina dedicada para aplicações de engenharia*. São Carlos: USP/IFQSC, 1986. 379p. (Dissertação de Doutorado).

[TRAINA_88] TRAINA Jr, C. e SLAETS, J.W. *Um modelo de representação de objetos*. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, 3, Recife, março 1988, p227-242.

Anais.

[TRAINA_89] TRAINA, Jr.,; TAKAI, O.K. e CORRÊA, P.L.P. *Uma linguagem de acesso ao MRO*. In: USP/ICMSC, São Carlos, 1989. *Notas Internas*.

[TRAINA_91] TRAINA Jr, C. *GEO: Um sistema de base de dados orientado a objetos -Estado atual de desenvolvimento e implementação*. In: SIMPÓSIO BRASILEIRO DE BANCO DE DADOS, Manaus, maio 1991, 78-93p. *Anais*.

[VALÊNCIO_90] VALÊNCIO, C.R. *Editor genérico sensível à sintaxe com as estruturas sintáticas armazenadas em base de dados*. São Carlos: USP/ICMSC, 1990. 78p. (Mini-dissertação apresentada para exame de qualificação para o Mestrado).

APÊNDICE 1

Comandos LAMRO que definem o esquema da empresa de engenharia

Especificação completa, através dos comandos na Linguagem de Acesso ao MRO, do esquema que modela uma empresa de engenharia (Figura 6), utilizado como exemplo durante a dissertação.

```
DEFINE OBJETO Fornecedor;
DEFINE OBJETO Peca;
DEFINE OBJETO Equipamento;
DEFINE OBJETO Projeto;
DEFINE OBJETO Funcionario;
DEFINE SUBTIPO OBJETO Administrativo
    TIPO MESTRE Fucionario
    PREDICADO Funcao == "administrativo"
    INICIALIZACAO
        ATRIBUTO Funcao "administrativo";
DEFINE SUBTIPO OBJETO Tecnico
    TIPO MESTRE Fucionario
    PREDICADO Funcao == "tecnico"
    INICIALIZACAO
        ATRIBUTO Funcao "tecnico"
        ATRIBUTO Especialidade "programador";
DEFINE COLONIA PROJETO
    CONSTRITA POR Projeto
    HABITADA POR Equipamento;
DEFINE RELACIONAMENTO Utiliza
    OPOSTO Utilizado
    ORIGEM Projeto
    DESTINO Equipamento;
DEFINE RELACIONAMENTO Participa
    OPOSTO Participam
    ORIGEM Funcionario
    DESTINO Projeto
    MINIMO 1;
DEFINE RELACIONAMENTO Usa
    OPOSTO Usada
    ORIGEM Equipamemento
    DESTINO Peca
    MINIMO 1
    MINIMO OPOSTO 1;
DEFINE RELACIONAMENTO Fornece
```

```
OPOSTO Fornecida
  ORIGEM Fornecedor
  DESTINO Peca
  MINIMO 1
  MINIMO OPOSTO 1;
DEFINE RELACIONAMENTO TRIPLO Fornece_e_Usa
  MINIMO 1
  MODALIDADE UM Fornecida CORRESPONDENTE Sido_Fornecida
  MODALIDADE DOIS Usada CORRESPONDENTE Para_Usar_No
  MINIMO 1
  DIVISAO UM Peca
  DIVISAO DOIS Fornecedor
  DIVISAO TRES Equipamento ;
DEFINE COMENTARIO Descricao ASSOCIADO A OBJETO Projeto;
DEFINE TEMPO Data_de_Inicio
  TIPO DE DADO Data
  ASSOCIADO A
    RELACIONAMENTO Participa Funcionario Projeto
    OBJETO Projeto
  MINIMO 1
  MAXIMO 1;
DEFINE SINONIMO cic ASSOCIADO A OBJETO Funcionario;
DEFINE PROPRIEDADE Salario
  TIPO DE DADO Real
  ASSOCIADO A OBJETO Pessoa;
DEFINE SINONIMO Nro_Proceso ASSOCIADO A OBJETO Projeto;
DEFINE PROPRIEDADE Valor
  TIPO DE DADO Real
  ASSOCIADO A OBJETO Equipamento;
DEFINE TEMPO Data_Fornecimento
  TIPO DE DADO DATA
  ASSOCIADO A
    RELACIONAMENTO Fornece Fornecedor Peca;
DEFINE PROPRIEDADE Setor
  ASSOCIADO A OBJETO Administrativo
  MINIMO 1;
DEFINE PROPRIEDADE Especialidade
  ASSOCIADO A OBJETO Tecnico
  MINIMO 1;
DEFINE TEMPO Data_Inicio
  TIPO DE DADO DATA
  ASSOCIADO A
    OBJETO Projeto
    RELACIONAMENTO Participa Funcionario Projeto
  MINIMO 1
  MAXIMO 1;
```

APÊNDICE 2

DRO do Meta-Esquema do MRO

APÊNDICE 3

BNF da Linguagem de Acesso ao MRO

```
<lamro> ::= /* vazio */
           | <lamro> <comandos>

<comandos> ::= <comando> ';'
              | <comandos> <comando> ';'

<comando> ::= <comando_esquema>
              | <comando_base>
              | <comando_ambiente>
              | <comando_extensao>

              /* comandos de extensao */

<comando_extensao> ::= INICIA
                      | HISTORIA
                      | '!' <numeros>
                      | '!' '!'
                      | LIGA <cad_str_varias>
                      | MANUAL <cad_str_varias>
                      | MANUAL

<comando_ambiente> ::= NOME <cad_str> USUARIO <cad_str>
                      CODIGO <cad_str>
                      | REQUISITA <cad_str> <cad_str>
                        <cad_str> para_ler_escrever
                      | LIBERA <cad_str> <cad_str> <cad_str>
                      | INSTANCIA <cad_str> <cad_str> <cad_str> <cad_str>
                      | CONGELA <cad_str> <cad_str> <cad_str> <cad_str>
                      | VARIANTE <cad_str> <cad_str> <cad_str>
                      | RESTAURA
                      | GRAVA
                      | FINALIZA
                      | FIM

                      /* Comandos que manipulam a colonia BEsquema */

<comando_esquema> ::= <inc_eli_de_t_de_obj>
                      | <inc_eli_de_t_de_col>
                      | <inc_eli_de_t_de_rel>
                      | <inc_de_t_de_tri>
                      | <eli_de_t_de_tri>
                      | <inc_eli_de_t_de_atr>
                      | <inc_eli_de_sb_de_obj>
```

```

| <inc_eli_de_sp_de_obj>
| <inc_eli_de_t_de_mas>
| <inc_eli_de_t_de_sel>
| <sub_de_t_de_col>
| <sub_de_sb_de_obj>
| <sub_de_seg_mul>
| <mos_t_ele>

<inc_eli_de_t_de_obj> ::= <def_eli> OBJETO <cad_str_varias>
                           <cl_identificado>
<cl_identificado>: /* vazio */
                  | IDENTIFICADO <t_relacionamento>

<inc_eli_de_t_de_col> ::= DEFINE COLONIA <cad_str>
                           CONSTRITA <cad_str> <cl_habitada>
                           | ELIMINA TIPO COLONIA <cad_str_varias>

<cl_habitada> ::= /* vazio */
                | HABITADA <cad_str_varias>

<inc_eli_de_t_de_rel> ::= <def_eli> RELACIONAMENTO <cad_str>
                           <cl_oposto> <cl_ts_de_rels>

<cl_oposto> ::= /* vazio */
              | OPOSTO <cad_str>

<cl_ts_de_rels> ::= <cl_t_de_rel>
                  | <cl_ts_de_rels> <cl_t_de_rel>

<cl_t_de_rel> ::= ORIGEM <cad_str> DESTINO <cad_str>
                  <cl_seguranca> <cl_minimo> <cl_maximo>
                  <cl_minimo_op> <cl_maximo_op>

<cl_seguranca> ::= /* vazio */
                  | <seguranca>

<cl_minimo> ::= /*vazio*/
               | <minimo>

<cl_maximo> ::= /* vazio */
               | <maximo>

<cl_minimo_op> ::= /* vazio */
                 | MINIMO OPOSTO <numeros>

<cl_maximo_op> ::= /* vazio */
                 | MAXIMO OPOSTO <numeros>

<inc_de_t_de_tri> ::= DEFINE TRIPLO <cad_str>
                      <cl_minimo> <cl_maximo> <cl_seguranca>
                      MODALIDADE UM <cad_str>
                      CORRESPONDENTE <cad_str>

```

```

                                <cl_minimo> <cl_maximo>
                                <cl_seguranca>
MODALIDADE DOIS <cad_str>
                                CORRESPONDENTE <cad_str>
                                <cl_divisoes>

<eli_de_t_de_tri>::= ELIMINA TIPO TRIPLO <cad_str> <cl_divisoes>

<cl_divisoes>::= <cl_divisao>
                | <cl_divisoes> <cl_divisao>

<cl_divisao>::= DIVISAO UM <cad_str>
                DIVISAO DOIS <cad_str_varias>
                DIVISAO TRES <cad_str>

<inc_eli_de_t_de_atr>::= <def_eli> <caracteristica> <cad_str>
                        <cl_t_de_dado> <cl_valor>
                        <cl_associacao>

<cl_t_de_dado>::= /* vazio */
                | TIPO DADO <cad_str>

<cl_valor>::= /* vazio */
                | VALOR <cad_str_varias>

cl_associacao::= /* vazio */
                | ASSOCIADO <cl_elemento>
                  <cl_minimo> <cl_maximo>

<cl_elemento>::= OBJETO <cad_str_varias>
                | RELACIONAMENTO <ts_relacionamentos>
                | TRIPLO <ts_rels_triplos>

<inc_eli_de_sb_de_obj>::= <def_eli> SUBTIPO <elemento>
                        <cad_str> TIPO MESTRE
                        <t_objeto_ou_t_relacionamento>
                        PREDICADO <cad_str>
                        INICIALIZACAO <cad_str_varias>

<inc_eli_de_sp_de_obj>::= <def_eli> SUPERTIPO <cad_str>
                        COLONIA <cad_str>

inc_eli_de_t_de_mas::= <def_eli> <cl_mascara>

<cl_msacara>::= MASCARA <cad_str> '{' <cad_str_varias> '}'
                <cl_ordem> <cl_repeticao>

<cl_ordem>::= /* vazio */
                | ORDEM <cresc_decresc>

<cl_repeticao>::= /* vazio */
                | REPETICAO <sim_nao>

```

```

<inc_eli_de_t_de_sel>::= <def_eli> SELECAO <cad_str_opc>

/* Comandos de substituicao */

<sub_de_seg_mul>::= SUBSTITUI <seg_mul> <cl_rel_atr_tri> PARA
    <seg_min_max>

<seg_mul>::= SEGURANCA
    | MULTIPLICIDADE

<cl_rel_atr_tri>::= ATRIBUTO <cad_str>
    | RELACIONAMENTO <t_relacionamento>
    | TRIPLO <t_rel_triplo>

<seg_min_max>::= <minimo> <maximo>
    | <seguranca>

<mos_t_ele>::= MOSTRA <especies> <elemento> <cad_str_opc>

/* Comandos na base de dados propriamente dita */

<comando_base>::= <inc_eli_loc_de_elem>
    | <inc_exc_de_sel>
    | <loc_de_cada_sel>
    | <mos_ele>
    | <lis_sel>

<inc_eli_loc_de_elem>::= <operacao> <elemento>
    <cad_str_varias>

<inc_exc_de_sel>::= ADICIONA <selecao>
    | EXCLUI <selecao>

<selecao>::= <corrente_opc>
    | '@'
    | <cad_str> <cad_str_varias>
    | TIPO <cad_str>
    | HABITAM <cad_str>
    | SELECAO <cad_str>
    | <selecao> '+' <selecao>
    | <selecao> '-' <selecao>
    | <selecao> '*' <selecao>
    | <relacionamento_selecao>
    | <selecao> ATRIBUTO <expressao>
    | <selecao> QUE_TEM ATRIBUTO <cad_str>
    | <selecao> NEG ATRIBUTO <cad_str>
    | '(' <selecao> ')'

<relacionamento_selecao>::= <selecao> RELACIONAMENTO <cad_str>
    <selecao>
    | <selecao> DESTINO RELACIONAMENTO
    <cad_str> <selecao>

```



```

| <selecao> QUE_TEM RELACIONAMENTO
  <cad_str>
| <selecao> DESTINO QUE_TEM
  RELACIONAMENTO <cad_str>
| <selecao> NEG RELACIONAMENTO <cad_str>
  <cad_str_sel>
| <selecao> DESTINO NEG RELACIONAMENTO
  <cad_str> <selecao>

<cad_str_sel> ::= /* vazio */
  | <selecao>

<loc_de_cada_sel> ::= LOCALIZA CADA <selecao> '{' comandos '}'

<sub_de_t_de_col> ::= SUBSTITUI COLONIA <cad_str> CONSTRITA
  <cad_str>

<sub_de_sb_de_obj> ::= SUBSTITUI SUBTIPO <cad_str> PREDICADO
  <cad_str>

<mos_ele> ::= MOSTRA <elemento> <cad_str_varias>

<lis_sel>: LISTA <selecao> <cl_mascara_opc>

<cl_mascara_opc> ::= /* vazio */
  | cl_mascara

<expressao> ::= <cad_str>
  | <cad_str> EQ <valor>
  | <cad_str> LT <valor>
  | <cad_str> LE <valor>
  | <cad_str> GE <valor>
  | <cad_str> GT <valor>
  | <cad_str> NE <valor>
  | expressao AND expressao
  | expressao OR expressao
  | NEG <expressao>
  | '(' expressao ')'

/* Comandos gerais */

<seguranca> ::= SEGURANCA <sim_nao>

<minimo> ::= MINIMO <numeros>

<maximo> ::= MAXIMO <numeros>

<elemento> ::= /* vazio */
  | OBJETO
  | COLONIA
  | RELACIONAMENTO
  | TRIPLO

```

```

| ATRIBUTO
| OBJETOS
| COLONIAS
| RELACIONAMENTOS
| TRIPLOS
| ATRIBUTOS
| <caracteristica>

<especies>::= TIPO
| SUBTIPO
| SUPERTIPO
| TIPOS
| SUBTIPOS
| SUPERTIPOS

<operacao>::= INCLUI
| LOCALIZA
| ELIMINA
| CONSIDERA

corrente_opc::= '.'
| CORRENTE

<para_ler_escrever>::= /* vazio */
| ESCREVER
| LER

<t_relacionamento>::= <cad_str> <cad_str> <cad_str>

<ts_relacionamentos>::= <t_relacionamento>
| <ts_relacionamentos> <t_relacionamento>

<t_rel_triplo>::= <cad_str> <cad_str> <cad_str> <cad_str>

<ts_rels_triplos>::= <t_rel_triplo>
| <ts_rels_triplos> <t_rel_triplo>

<t_objeto_ou_t_relacionamento>::= <cad_str> <ori_des_opc>

<ori_des_opc>::= /* vazio */
| <cad_str> <cad_str>

<sim_nao>::= SIM
| NAO

<def_eli>::= DEFINE
| ELIMINA TIPO

<cresc_decresc>::= CRESCENTE
| DECRESCENTE

<caracteristica>::= PROPRIEDADE
```

```

| SINONIMO
| COMENTARIO
| GRAFICO
| TEMPO
| IMAGEM
| VISUALIZACAO
| PROCEDIMENTO

```

```

<cad_str> ::= <nome>
| <string>
| <numeros>
| '.'

```

```

<cad_str_opc> ::= /* vazio */
| <cad_str>

```

```

<cad_str_varias> ::= <cad_str>
| <cad_str_varias> <cad_str>

```

```

<nome> ::= <letra> <numeros_letras>

```

```

<numeros_letras> ::= /* vazio */
| <numeros_letras> <numero_letra>
| '_'

```

```

<numero_letra> ::= <decimal>
| <letra>

```

```

<numeros> ::= '.' <decimais>
| <decimais> '.' <decimais>
| <decimais> '.'

```

```

<decimais> ::= <decimal>
| <decimais> <decimal>

```

```

<decimal> ::= '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

```

```

<letra> ::=
'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m'|'n'|
'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'x'|'z'|'A'|'B'|'C'|'D'|
'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M'|'N'|'O'|'P'|'Q'|'R'|
'S'|'T'|'U'|'V'|'X'|'Z'

```

```

<character> ::= /* conjunto de caracteres ASCII de 0 a 255 */

```

```

<string> ::= '"' <caracteres> '"'

```

```

<caracteres> ::= <character>
| <caracteres> <character>

```

APÊNDICE 4

Conjunto de palavras reservadas da gramática

ATRIBUTO	ATRIBUTOS	ASSOCIADO	CADA
CARACTERISTICA	CODIGO	DADO	COLONIA
COLONIAS	CONGELA	CONSTRITA	CONSIDERA
CONSTRITO	CORRENTE	CORRENTES	CORRESPONDENTE
CRESCENTE	DECRESCENTE	DEFINE	DESTINO
DIVISAO	DOIS	DIVISAO	ELIMINA
EXCLUI	FIM	FINALIZA	GRAVA
HABITADA	HABITAM	HISTORIA	INCLUI
IDENTIFICA	IDENTIFICADO	INICIA	INICIALIZACAO
INSTANCIA	ADICIONA	LER	LIGA
LISTA	LIBERA	MANUAL	LOCALIZA
MASCARA	MAXIMO	MESTRE	MINIMO
MODALIDADE	MOSTRA	NAO	NOME
OBJETO	OBJETOS	OPOSTO	ORIGEM
PREDICADO	QUEM	RELACIONAMENTO	RELACIONAMENTOS
REQUISITA	REPETICAO	RESTAURA	SEGURANCA
SELECAO	SIM	SUBSTITUI	SUBTIPO
SUBTIPOS	SUPERTIPO	SUPERTIPOS	TIPO
TIPOS	TRES	TRIPLO	TRIPLOS
UM	USUARIO	VALOR	VARIANTE

APÊNDICE 5

Especificação da gramática para o YACC

%token	ORDEM	REPETICAO	NOME	USUARIO
	CODIGO	FIM	MODALIDADE	UM
	VALOR	SIM	SUBTIPOS	NAO
	ATRIBUTOS	ASSOCIADO	ATRIBUTO	CADA
	CARACTERISTICA	DADO	COLONIA	COLONIAS
	CONGELA	CONSTRITA	CONSIDERA	CONSTRITO
	CORRENTE	CORRENTES	CORRESPONDENTE	CRESCENTE
	DECRESCENTE	DEFINE	DESTINO	DIVISAO
	DOIS	ESCREVER	ELIMINA	EXCLUI
	FINALIZA	GRAVA	HABITADA	HABITAM
	HISTORIA	INCLUI	IDENTIFICA	IDENTIFICADO
	INICIA	INICIALIZACAO	INSTANCIA	ADICIONA
	LER	LIGA	LISTA	LIBERA
	MANUAL	LOCALIZA	MASCARA	MAXIMO
	MESTRE	MINIMO	MOSTRA	OBJETO
	OBJETOS	OPOSTO	ORIGEM	PREDICADO
	QUEM	RELACIONAMENTO	RELACIONAMENTOS	REQUISITA
	RESTAURA	SEGURANCA	SELECAO	SUBSTITUI
	SUBTIPO	SUPERTIPO	SUPERTIPOS	TIPO
	TIPOS	TRES	TRIPLO	TRIPLOS
	VARIANTE	QUE_TEM	NEG	

```

%left '*' '/'
%left '+' '-'
/* %lef UMINU */ /* operador menos */
/* operadores relacionais:*/
%left EQ NE /* == != */
%left LT GT LE GE /* < > <= >= */
/* operadores logicos para expressao */
%left AND OR
/* operadores de selecao */
%left /* NEG_ATRIBUTO*/ ATRIBUTO
%left TIPO SELECAO SUPERTIPO CONSTRINGE HABITAM
%left DESTINO
%left /* NEG_RELACIONAMENTO */ RELACIONAMENTO
%left '^'
%left UNIAO DIFERENCA /* + - */
/* %LEFT EQ NE LT LE GT GE
%LEFT '+' '-' '^' */ /* left associative, same precedence */
%token <simbolo> CADEIA CHARACTER NUMERO STRING PONTO
%type <simbolo> cad_str cl_oposto cl_seguranca sim_nao
cl_t_de_dado cad_str_opc

```

```
%token <valor> OBJETO COLONIA RELACIONAMENTO TRIPLO ATRIBUTO
OBJETOS COLONIAS RELACIONAMENTOS TRIPLOS ATRIBUTOS
TIPO SUPERTIPO SUBTIPO INCLUI
TIPOS SUPERTIPOS SUBTIPOS CARACTERISTICA_T
ELIMINA LOCALIZA CONSIDERA
```

```
%type <valor> operacao especies elemento
para_ler_escrever def_eli caracteristica
selecao relacionamento selecao cad_str_sel
comando_extensao comando
```

```
%%
```

```
/* *****
/* SINTAXE DO LAMRO QUE MANIPULA O ESQUEMA DA BASE MRO */
/* *****
```

```
lamro: /* vazio */
      | lamro '\n'
      | lamro comando '\n'
      | error '\n'
```

```
;
comando: comando_esquema
        | comando_base
        | comando_ambiente
        | comando_extensao
```

```
;
comando_extensao : INICIA pto_vir
                  | HISTORIA pto_vir
                  | '!' NUMERO pto_vir
                  | '!' '!' pto_vir
                  | MANUAL cad_str_varias pto_vir
                  | MANUAL pto_vir
```

```
;
comando_ambiente: NOME cad_str USUARIO cad_str
                  CODIGO cad_str pto_vir
                  | REQUISITA cad_str CONSTRITA cad_str cad_str
para_ler_escrever pto_vir
                  | LIBERA cad_str CONSTRITA cad_str cad_str pto_vir
                  | INSTANCIA cad_str cad_str cad_str cad_str pto_vir
                  | CONGELA cad_str cad_str cad_str cad_str pto_vir
                  | VARIANTE cad_str cad_str cad_str pto_vir
                  | LIGA cad_str_varias pto_vir
                  | RESTAURA pto_vir
                  | GRAVA pto_vir
                  | FIM pto_vir
```

```
/* comandos que manipulam o esquema */
```

```
comando_esquema: inc_eli_de_t_de_obj
                  | inc_eli_de_t_de_col
                  | inc_eli_de_t_de_rel
                  | inc_de_t_de_tri
                  | inc_eli_de_t_de_atr
```

```

        | inc_eli_de_sb_de_obj
        | inc_eli_de_sp_de_obj
        | inc_eli_de_t_de_mas
        | inc_eli_de_t_de_sel
        | sub_de_t_de_col
        | sub_de_sb_de_obj
        | mos_t_ele
    ;
inc_eli_de_t_de_obj: def_eli OBJETO cad_str_varias {tip_rel=FALSE;}
                    cl_identificado pto_vir
;
cl_identificado: /* vazio */
                | IDENTIFICADO t_relacionamento
;
inc_eli_de_t_de_col: DEFINE COLONIA cad_str CONSTRITA cad_str
                    cl_habitada pto_vir
                    | ELIMINA TIPO COLONIA cad_str_varias
pto_vir
;
cl_habitada: /* vazio */
            | HABITADA cad_str_varias
;
inc_eli_de_t_de_rel: def_eli RELACIONAMENTO cad_str cl_oposto
cl_ts_de_rels
                    pto_vir
;
cl_oposto: /* vazio */
          | OPOSTO cad_str
;
cl_ts_de_rels: cl_t_de_rel
              | cl_ts_de_rels cl_t_de_rel
;
cl_t_de_rel: ORIGEM cad_str DESTINO cad_str cl_seguranca cl_minimo
cl_maximo
            cl_minimo_op cl_maximo_op
;
cl_seguranca: /* vazio */
             | SEGURANCA sim_nao
;
cl_minimo: /*vazio*/
          | MINIMO NUMERO
;
cl_maximo: /* vazio */
          | MAXIMO NUMERO
;
cl_minimo_op: /* vazio */
             | MINIMO OPOSTO NUMERO
;
cl_maximo_op: /* vazio */
             | MAXIMO OPOSTO NUMERO
;
inc_de_t_de_tri: DEFINE TRIPLO cad_str

```

```

        cl_minimo cl_maximo cl_seguranca
        MODALIDADE UM cad_str
            CORRESPONDENTE cad_str
            cl_minimo cl_maximo cl_seguranca
        MODALIDADE DOIS cad_str
            CORRESPONDENTE cad_str
            cl_divisoos pto_vir
    ;
cl_divisoos: cl_divisao
    | cl_divisoos cl_divisao
    ;
cl_divisao: DIVISAO UM cad_str DIVISAO DOIS cad_str_varias
    DIVISAO TRES cad_str
    ;
inc_eli_de_t_de_atr: def_eli caracteristica cad_str cl_t_de_dado
cl_valor
    cl_associacao pto_vir
    ;
cl_t_de_dado: /* vazio */
    | TIPO DADO cad_str
    ;
cl_valor: /* vazio */
    | VALOR cad_str_varias
    ;
cl_associacao: /* vazio */
    | ASSOCIADO
        cl_elemento cl_minimo cl_maximo
    ;
cl_elemento: OBJETO cad_str_varias
    | RELACIONAMENTO ts_relacionamentos
    | TRIPLO ts_rels_triplos
    ;
inc_eli_de_sb_de_obj: def_eli SUBTIPO elemento cad_str TIPO MESTRE
    t_objeto_ou_t_relacionamento PREDICADO cad_str
    INICIALIZACAO cl_atributos pto_vir
    ;
cl_atributos: cl_atributo
    | cl_atributos cl_atributo
    ;
cl_atributo: ATRIBUTO cad_str cad_str
    ;
inc_eli_de_sp_de_obj: def_eli SUPERTIPO cad_str COLONIA cad_str
pto_vir
    ;
inc_eli_de_t_de_mas: def_eli MASCARA cad_str ATRIBUTO
cad_str_varias
    cl_ordem cl_repeticao pto_vir
    ;
cl_ordem: /* vazio */
    | ORDEM_cresc_decresc cad_str
    ;
cl_repeticao: /* vazio */

```



```

        | REPETICAO sim_nao
    ;
inc_eli_de_t_de_sel: def_eli SELECAO cad_str_opc pto_vir
    ;
mos_t_ele: MOSTRA especies elemento cad_str_opc pto_vir /*
tipo_de_obj nome_obj */

/* comandos que manipulam a base de dados */

comando_base: inc_eli_loc_de_elem
    | inc_exc_de_sel
    | loc_de_cada_sel
    | mos_ele
    | lis_sel
    ;

/* definicoes das instancias dos componentes do modelo */

inc_eli_loc_de_elem: operacao elemento cad_str_varias pto_vir

/* manipulacao de selecoes de objeto */
inc_exc_de_sel: ADICIONA selecao pto_vir
    | EXCLUI selecao pto_vir
    ;
selecao: corrente_opc
    | '@'
    | cad_str cad_str_varias
    | TIPO cad_str
    | HABITAM cad_str
    | CONSTRINGE COLONIAS cad_str
    | SUPERTIPO cad_str cad_str cad_str
    | SELECAO cad_str
    | selecao '+' selecao %prec UNIAO
    | selecao '-' selecao %prec DIFERENCA
    | selecao '*' selecao
    | relacionamento selecao
    | selecao QUE_TEM ATRIBUTO cad_str %prec ATRIBUTO
    | selecao NEG ATRIBUTO cad_str %prec ATRIBUTO
    | '(' selecao ')' { $$ = $2;}
    ;
relacionamento_selecao: selecao RELACIONAMENTO cad_str selecao
    | selecao DESTINO RELACIONAMENTO cad_str
    | selecao QUE_TEM RELACIONAMENTO cad_str
    | selecao DESTINO QUE_TEM RELACIONAMENTO
    | selecao NEG RELACIONAMENTO cad_str
    | selecao DESTINO NEG RELACIONAMENTO cad_str
    ;

```

```
cad_str_sel: /* vazio */
            | selecao
            ;
loc_de_cada_sel: LOCALIZA CADA selecao
                '{' comandos '}'
            ;
sub_de_t_de_col: SUBSTITUI COLONIA cad_str CONSTRITA cad_str
pto_vir
            ;
sub_de_sb_de_obj: SUBSTITUI SUBTIPO cad_str PREDICADO cad_str
pto_vir
            ;
mos_ele: MOSTRA elemento cad_str_varias pto_vir
            ;
lis_sel: LISTA selecao MASCARA cad_str pto_vir
            ;

            /***** sentencas genericas *****/

elemento: /* vazio */
        | OBJETO
        | COLONIA
        | RELACIONAMENTO
        | TRIPLO
        | ATRIBUTO
        | OBJETOS
        | COLONIAS
        | RELACIONAMENTOS
        | TRIPLOS
        | ATRIBUTOS
        | caracteristica
        ;
especies: TIPO
        | SUBTIPO
        | SUPERTIPO
        | TIPOS
        | SUBTIPOS
        | SUPERTIPOS
        ;
operacao: INCLUI
        | LOCALIZA
        | ELIMINA
        | CONSIDERA
        ;

corrente_opc: PONTO
            | CORRENTE
            ;
pto_vir: ';'
            ;
para_ler_escrever: /* vazio */
                | ESCREVER
```

```
        | LER
    ;
t_relacionamento: cad_str cad_str cad_str
    ;
ts_relacionamentos: t_relacionamento
    | ts_relacionamentos t_relacionamento
    ;
t_rel_triplo: cad_str cad_str cad_str cad_str
    ;
ts_rels_triplos: t_rel_triplo
    | ts_rels_triplos t_rel_triplo
    ;
t_objeto_ou_t_relacionamento: cad_str ori_des_opc
    ;
ori_des_opc: /* vazio */
    | cad_str cad_str
    ;
sim_nao: SIM
    | NAO
    ;
def_eli: DEFINE
    | ELIMINA TIPO
    ;
cresc_decresc: CRESCENTE
    | DECRESCENTE
    ;
caracteristica: CARACTERISTICA_T
    ;
cad_str: CADEIA
    | STRING
    | NUMERO
    | PONTO
    ;
cad_str_opc: /* vazio */
    | cad_str
    ;
cad_str_varias: cad_str
    | cad_str_varias cad_str
    ;
%%      /* fim da sintaxe */
```