UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM ESQUEMA COMPILATIVO PARA LINGUAGENS DE OPERAÇÃO DE BANCO DE DADOS

por

CIRANO IOCHPE

Dissertação submetida como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profª Lia Goldstein Golendziner
Orientadora

Porto Alegre, fevereiro de 1984

CATALOGAÇÃO NA FONTE

Iochpe, Cirano

Um esquema compilativo para linguagens de operação de banco de dados. Porto Alegre, PGCC da UFRGS, 1984.

1v.

Diss. (mestr.ci.comp.) UFRGS-PGCC, Porto Alegre, BR-RS, 1983.

Dissertação: LOBAN: Sistema L :
Sistemas de Gerência de bancos de dados:
Sistemas Compilativos: Geração de Código

À Eliete,
companheira de sempre

e

Aos Meus Pais
que desde cedo me transmitiram o
gosto pelo saber

AGRADECIMENTOS

À profª Lia Goldstein Golendziner, pelo estímulo e dedicação constantes, na orientação de todas as etapas deste trabalho.

Ao prof. Carlos Alberto Heuser, pelos conhecimentos introdutórios em banco de dados e co-orientação.

Ao Centro de Processamento de Dados da Universidade Federal do Rio Grande do Sul, na pessoa de seu diretor, prof.Edemundo da Rocha Vieira, pelo apoio e incentivo durante todo o Curso de Pós-Graduação.

Ao prof. Francisco Schlabitz, pela motivação inicial ao estudo da Ciência da Computação.

Ao Dr. Bernardo Brunstein cujo apoio foi de grande valia para a concretização deste trabalho.

Ao Curso de Pós-Graduação em Ciência da Computação, na pessoa de seu coordenador, prof. Philippe O. A. Navaux e à IBM do Brasil Ltda., pelo apoio financeiro.

Ao pessoal da Biblioteca do Centro de Processamento de Dados, pelo apoio e auxílio prestados.

À Maria Helena Amaral Cardozo, pela datilografia, e a George Augusto M. de Moraes, pelos desenhos.

A todos aqueles que, de alguma forma, vieram a contribuir para a realização deste trabalho.

SUMÁRIO

LISTA DE TABELAS

LISTA DE FIGURAS

RESUMO

Este trabalho apresenta o estudo feito sobre esquemas compilativos para linguagens não-procedurais, de operação de banco de dados, baseados na abordagem relacional e propõe uma técnica de geração de código para a linguagem LOBAN (Linguagem de Operação de Banco de Dados), sendo implementada, através do Sistema L, na Universidade Federal do Rio Grande do Sul. LOBAN é uma linguagem de alto nível que apresenta, entre outras, funções equivalentes às da álgebra relacional.

São apresentados o projeto e a especificação de um módulo gerador de código para o Sistema L, baseado na técnica proposta, além de uma nova arquitetura para o sistema, a fim de que este suporte uma interface LOBAN embutida em Pascal.

ABSTRACT

This work presents a study of compilation schemes for non-procedural languages used as interfaces of database systems based on the relational approach, and proposes a code generation technique for LOBAN (in English, DABOL-Data Base Operation Language) now being implemented through System L, in the Universidade Federal do Rio Grande do Sul (Brazil). DABOL is a high level language that offers, among others, functions equivalents to those of the relational algebra.

The project and specification of a code generation module for the System L, based on the proposed technique, are presented together with a new architecture of the system that supports an embedded interface of LOBAN in Pascal.

1. INTRODUÇÃO

1.1 Motivação

A característica principal das linguagens não-procedurais é permitir ao usuário manipular seus dados,sem ter conhecimento de sua estrutura interna nem da ordem em que as operações, por ele especificadas, irão ocorrer. Exemplos de linguagens não-procedurais são aquelas baseadas em álgebra ou cálculo relacional, como SQL [IBM 81], QUEL [STO 76] e LOBAN [SAN 80].

Os sistemas de gerência de bancos de dados baseados na abordagem relacional, na sua maioria, oferecem linguagens não-procedurais. O modelo conceitual destes sistemas é bastante simples (os dados são organizados em tabelas) e o usuário dispõe de um conjunto poderoso de instruções, orientadas à manipulação de conjuntos de dados, de fácil entendimento e aplicação.

Devido à grande carga semântica das instruções de linguagens não-procedurais utilizadas em sistemas de banco de dados, parece ser consenso entre os implementadores, que esquemas compilativos representam um aumento real de eficiência em relação à interpretação das instruções, em tempo de execução [STO 80], [KAT 79], [LOR 79].

Os compiladores para linguagens não-procedurais devem executar uma série de funções inexistentes nos esquemas compilativos para linguagens de alto nível convencionais. Algumas destas funções são:

- decisão da ordem de execução das operações submetidas, ao sistema, pelo usuário;

- escolha dos algoritmos mais eficientes para a execução de cada operação;

- implementação de "laços" de programa para reali

zar operações que manipulam conjuntos de dados.

Dentro do projeto MINIBAN, está sendo realizada a implementação de um subconjunto significativo da linguagem LOBAN. Esta implementação,denominada Sistema L, foi inicialmente pensado como um sistema interpretativo que geraria,a partir da linguagem LOBAN, um código executável por uma máquina virtual, que conta com um sistema operacional dedicado (SOL - Sistema Operacional para LOBAN). Entretanto, através de estudos realizados no decorrer do projeto e que ratificam experiências adquiridas em outros sistemas ([LOR 79], [STO 80], [CHA 81]), concluiu-se que um esquema compilativo poderia ser mais eficiente, além de exigir um esforço de implementação menor em sistemas que utilizam linguagens não-procedurais.

Decidiu-se então modificar a arquitetura do Sistema L, tornando-o um sistema compilativo. As características específicas da proposta original, que contribuíram para esta decisão, são:

- o espaço de memória, exigido por um interpretador, é relativamente grande para um minicomputador, no caso o LABO 8034 sobre o qual está sendo realizada a implementação;

- o interpretador proposto para o Sistema L, assim como um compilador,executa passos de tradução, gerando códigos intermediários que, no entanto, não são reaproveitados nas próximas execuções de uma mesma consulta.

Outra decisão importante, tomada a partir dos estudos realizados, foi a de tornar a linguagem LOBAN uma interface de linguagem hospedeira, no que diz respeito à manipulação de dados, mantendo-se a interface autônoma apenas para as funções de administração da base de dados, tais como definição dos dados e reconstrução. Devido às características apresentadas pela máquina alvo (a implementação está sendo feita sobre o sistema Pascal Concorrente [BRI 77]), a

linguagem hospedeira escolhida foi o Pascal. Com a criação desta nova interface, a implementação do sistema fica facilitada, pois funções como entrada/saída de dados e avaliação de expressões serão realizadas pelas instruções Pascal e não mais pelas de LOBAN.

Este trabalho tem, por objetivo, estudar as possibilidades de esquemas compilativos de linguagens não-procedurais para acesso a bancos de dados baseados na abordagem relacional e, levando em conta as partes do Sistema L que já foram projetadas (analisadores léxico, sintático e semântico; otimizador algébrico e seletor de caminhos de acesso), propor:

a) uma nova arquitetura, do Sistema L, que suporte um esquema compilativo;

b) a arquitetura do pré-processador da linguagem LOBAN embutida em Pascal;

c) a especificação e o projeto de um módulo gerador de código, para o pré-processador, incluindo a definição do código a ser gerado.

1.2 Organização da dissertação

A dissertação apresentada é composta de 10 capítulos. O capítulo 1 visa introduzir o leitor aos objetivos do estudo realizado, incluindo, também, a organização da dissertação.

O capítulo 2 apresenta a Interface LOBAN descrevendo a estrutura da base de dados e as instruções da linguagem que formam o sub-conjunto a ser implementado pela interface embutida do sistema. Neste capítulo, são descritos os operadores LOBAN de manipulação de estruturas de dados bem como o mecanismo oferecido para endereçar estas estruturas.

O capítulo 3 apresenta a arquitetura antiga do Sistema L, mostrando seu esquema de comunicação com o usuário e descrevendo o funcionamento de cada uma de suas partes.

O capítulo 4 descreve o ambiente de implementação do sistema em termos de suas características de hardware e software. Aqui também são apresentadas as limitações deste ambiente que devem ser levadas em conta no projeto de implementação.

O capítulo 5 descreve, com maior profundidade, o estudo realizado sobre a arquitetura antiga do sistema e apresenta suas conclusões como justificativas para a proposta de uma nova arquitetura para o Sistema L.

O capítulo 6 apresenta um estudo detalhado de diferentes esquemas compilativos para linguagens não-procedurais de manipulação de bancos de dados, baseados na abordagem relacional. São apresentados aqui, também, exemplos de sistemas de gerência de banco de dados que implementaram alguns destes esquemas.

O capítulo 7 descreve a proposta da nova arquitetura para o Sistema L, apresentando o novo esquema de comunicação entre o usuário e o sistema,além da estrutura e dos módulos que implementam a interface LOBAN embutida. A partir disto, o esquema compilativo adotado é descrito.

O capítulo 8 apresenta a técnica de geração de código, projetada para linguagens não-procedurais, que será implementada no Sistema L. Neste capítulo, descreve-se a estrutura do código de entrada e a arquitetura do módulo gerador de código.

O capítulo 9 apresenta o projeto de implementação do módulo gerador de código do Sistema L, em detalhes. Neste capítulo, são apresentados, entre outras coisas, a ar

quitetura deste módulo para implementação, os algoritmos de funcionamento de cada uma de suas partes, as limitações que o módulo causa ao usuário e as modificações no ambiente propostas para aumentar sua eficiência.

O capítulo 10 apresenta observações feitas durante o projeto e conclusões sobre as decisões nele tomadas.

2. INTERFACE LOBAN

2.1 Introdução

A linguagem LOBAN (Linguagem de Operação de BANco de dados) foi definida durante a execução da primeira etapa do projeto MINIBAN, e sua descrição completa pode ser encontrada junto à bibliografia [RIC 78],[SAN 80].

LOBAN é a interface de comunicação entre o usuário e o sistema de gerência da base de dados, permitindo a descrição e a manipulação das informações contidas na base de dados e a execução de operações de entrada e saída.

Um subconjunto da linguagem está em implementação na Universidade Federal do Rio Grande do Sul, sob a denominação de Sistema L [HEU 83], [GOL 83]. Este capítulo descreve o subconjunto LOBAN escolhido para implementação.

2.2 A base de dados

A seguir é apresentada, de forma abstrata, a organização das informações da base de dados, como é vista pelo usuário, através da interface LOBAN. A descrição da base de dados será feita à luz dos conceitos de IMC (Information Management Concepts) desenvolvidos para representar sistemas de informações [DUR 76].

A *construção* é o conceito fundamental do IMC.Qualquer informação referenciável é considerada uma construção.

O conteúdo global da base de dados é denominado *acervo total*. Este acervo compõe-se de *acervos setoriais* (bancos de dados), identificáveis por seus nomes. Na prática, um acervo setorial constitui o ambiente de trabalho de uma aplicação. Os acervos setoriais não se relacionam entre si [GOL 82].

Os acervos setoriais contêm, entre outras construções, os arquivos relacionais que também podem ser identificáveis por seus nomes. A figura 2.1 apresenta um acervo setorial que será utilizado como exemplo no decorrer do texto.

Figura 2.1 - Exemplo de Acervo Setorial

Um arquivo relacional LOBAN (figura 2.2) é composto por uma tabela relacional (coleção de tuplas) sob o nome TR e, opcionalmente, uma tupla isolada, de nome FICHA, que mantém informações sobre o arquivo como um todo. Em analogia aos sistemas convencionais, a tabela relacional assemelhar-se-ia a um arquivo e, suas tuplas, aos registros deste arquivo.

Os arquivos relacionais, bem como seus componentes, podem ser manipulados por vários operadores que serão adiante descritos, inclusive por aqueles da álgebra relacional.

A linguagem LOBAN também oferece operadores que produzem tabelas ligacionais, as quais não podem ser mantidas na base de dados existindo, no máximo, até o fim da execução do programa que as cria. Estas construções são conjuntos de ligações, sendo cada ligação comparável a uma ocorrência de "SET" da abordagem CODASYL/DBTG. A figura 2.3 apresenta um exemplo de tabela ligacional onde, para cada

departamento da empresa, existe uma ligação composta por uma tupla com informações sobre o departamento (sob o nome L), e uma tabela relacional com informações sobre os empregados ali lotados (sob o nome T).

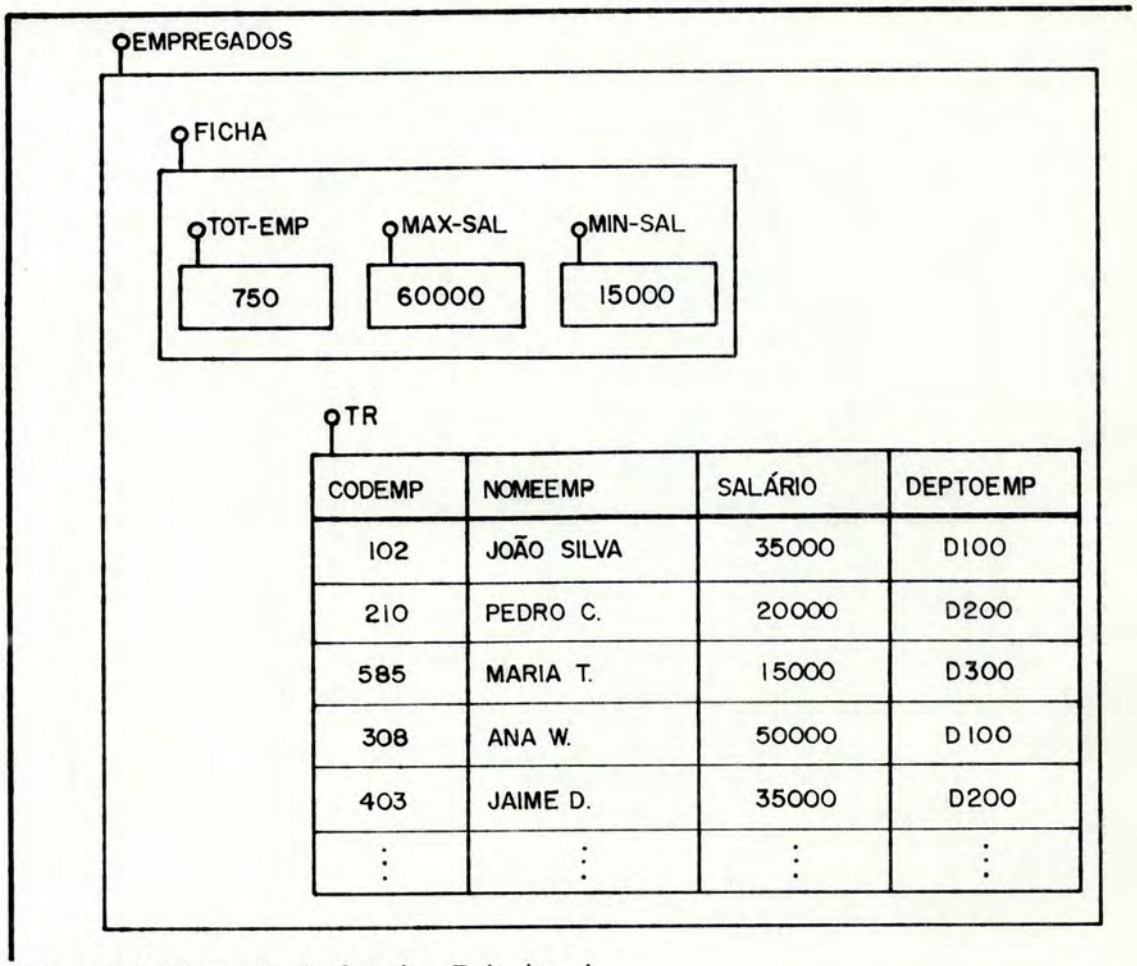

Figura 2.2 - Exemplo de Arquivo Relacional

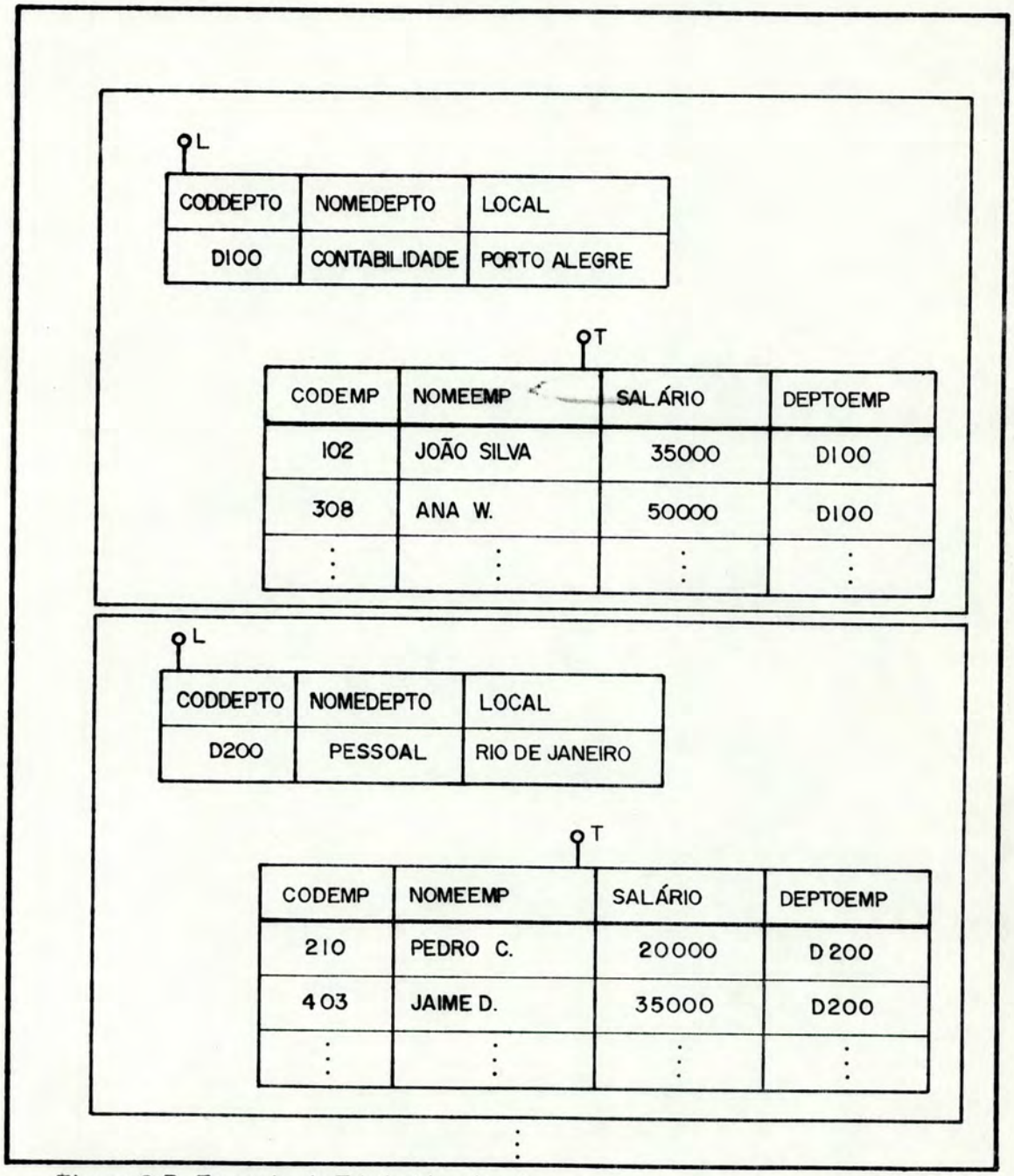

Figura 2.3 - Exemplo de Tabela Ligacional

2.3 As instruções LOBAN

2.3.1 Introdução

As instruções LOBAN estão classificadas em instruções administrativas e instruções operativas. As instruções administrativas permitem a definição dos dados, incluindo a especificação de restrições de integridade sobre os dados, de aspectos de controle de acesso e reconstrução da base de dados. Além disso estas instruções possibilitam a criação e deleção das estruturas de dados definidas. Também faz parte do grupo administrativo, a instrução de controle de fluxo (FAZER PARA CADA).

As instruções operativas realizam a representação de construções na saída (REPRESENTAR) e alterações em acervos setoriais (INCLUIR, EXCLUIR, SUBSTITUIR).

2.3.2 Sintaxe das instruções operativas e de controle de fluxo

A seguir, é apresentada a forma geral das instruções LOBAN operativas e de controle de fluxo, utilizando-se, para isso, a metalinguagem BNF com algumas extensões. Os metassímbolos que aparecem na descrição são apresentados na tabela 2.1.

```
<instruções operativas>::= <incluir>|<excluir>|<substituir>|
                           <representar>
<incluir>::= INCLUIR <expressão de obtenção de construções>
                     EM  <endereço de ponto>
<excluir>::= EXCLUIR DE  <endereço de ponto>
<substituir>::= SUBSTITUIR  <endereço de ponto>
                     POR <expressão de obtenção de construções>
<representar>::= REPRESENTAR <expressão de obtenção de construções>
                     EM  <identificador Pascal>
```

<fazer-para-cada>::= FAZER PARA CADA <endereço de ponto>
EM < expressão de obtenção de construções >
((instrução {LOBAN|Pascal})...)

METASSÍMBOLO	SIGNIFICADO
< >	IDENTIFICA METAVARIÁVEIS Ex: <x> É UMA METAVARIÁVEL
::=	DEFINIDA POR
|	OU (EXCLUSIVO)
[∝]	OPCIONAL: ∝ OU NADA
{ ∝ | β }	ALTERNATIVA: ∝ OU β
∝ ...	REPETIÇÃO: ∝ PODE APARECER UMA OU MAIS VEZES: ∝∝...∝
$\propto_{\underline{sss}}$	REPETIÇÃO: ∝ PODE APARECER UMA OU MAIS VEZES SENDO AS OCORRÊNCIAS SEPARADAS POR UM SEPARADOR ESPECÍFICO (REPRESENTADO AQUI POR "s")

Tabela 2.1- Metassímbolos utilizados na descrição de LOBAN

É nas expressões de obtenção de construções e no endereço de ponto que reside a potencialidade e flexibilidade das instruções LOBAN, sendo utilizadas tanto na manipulação dos dados como em sua definição.

2.3.3 Endereçamento de Pontos

A referência às construções, em LOBAN, é feita de forma única, conhecida como endereço de ponto [RIC 81], através da qual uma construção é identificada em um determinado contexto.

Um endereço de ponto tem a seguinte forma:

critério[.criterio]...

onde cada critério seleciona pontos em um nível de agregação de construções. Um critério pode ser, simplesmente, o nome da construção no ponto a selecionar (EMPREGADOS.TR para selecionar a tabela relacional do arquivo EMPREGADOS), ou uma expressão booleana complexa, que deverá ter valor verdadeiro para que o ponto seja referenciado (EMPREGADOS.TR.(C NOMEEMP = 'JOAO SILVA' OU C SALARIO < 35000) para selecionar a tupla do empregado João Silva e as tuplas de empregados cujos salários são menores que 35000, por exemplo).

As expressões booleanas de LOBAN podem ser formadas a partir de operadores relacionais (=,>,<,<>,<=,>=), lógicos (ET, OU, NÃO e IMPL), o operador booleano ELEM (testa se uma construção é elemento de uma coleção) e a constante VERDADEIRO.

No anexo 1, é apresentada a sintaxe completa de um endereço de ponto utilizando-se, para isto, a BNF estendida da tabela 2.1.

A seguir, são apresentados alguns exemplos de endereço de pontos, considerando os dados das figuras 2.2 e 2.3:

exemplo 1: endereçamento da construção de nome FICHA do arquivo relacional Empregados:

EMPREGADOS.FICHA

exemplo 2: endereçamento da tabela relacional do mesmo arquivo:

EMPREGADOS.TR

exemplo 3: referência a todas as tuplas de empregados que trabalham no departamento D200:

EMPREGADOS.TR.(C DEPTOEMP='D200')

no último critério aplicado neste exemplo, o operador C é usado para obter, em cada tupla de EMPREGADOS, o atributo DEPTOEMP que é, então, comparado com a cadeia de caracteres 'D200'.

exemplo 4: o endereço de ponto do exemplo anterior pode ser expandido a fim de referenciar apenas os nomes dos empregados que trabalham no departamento D200:

EMPREGADOS.TR.(C DEPTOEMP='D200').NOMEEMP

O conceito de endereço de ponto pode também ser aplicado a arquivos ligacionais, como mostram os exemplos abaixo:

exemplo 5: usando-se a tabela ligacional da figura 2.3 como contexto de referência, pode-se identificar a ligação do departamento de contabilidade e referenciar a construção sob nome LOCAL desta ligação:

(C L.NOMEDEPTO = 'CONTABILIDADE').L.LOCAL

Os primeiros dois critérios selecionam a ligação da tabela ligacional cujo atributo NOMEDEPTO da tupla de ligante (L) tem valor 'CONTABILIDADE'. Os terceiro e quarto critérios referenciam a construção sob o nome LOCAL (também da tupla de ligante) desta ligação.

exemplo 6: para endereçar as tuplas dos empregados do departamento de contabilidade que recebem salário maior que 40.000, pode-se utilizar o endereço de ponto abaixo:

```
(C L.NOMEDEPTO = 'CONTABILIDADE').T.(C SALARIO>40000)
```

É importante observar que o endereço de ponto não identifica, necessariamente, um único ponto, mas sim o conjunto dos pontos que satisfazem os critérios de referência.

2.3.4 Expressões de obtenção de construções

O resultado da execução destas expressões é uma construção que pode ser cópia de outra já existente na base de dados ou no canal auxiliar, ou pode ter sido obtida a partir de operações sobre outras construções.

As expressões de obtenção de construções aparecem, sempre, dentro de instruções operativas que operam sobre as construções obtidas da base de dados, canal auxiliar ou de fora do sistema.

Estas expressões utilizam-se de operadores que podem produzir, como resultado, tuplas, ligações, tabelas (relacionais ou ligacionais), coleções de itens ou valores numéricos.

A seguir, os operadores de obtenção de construções são descritos sintática e semanticamente, utilizando-se, novamente, a BNF estendida apresentada na tabela 2.1.Os exemplos que acompanham esta descrição baseiam-se nas estruturas de dados das figuras 2.2 e 2.3. É importante observar que os termos <obter tabela relacional> e <obter tabela ligacional> são utilizados para representar expressões de obtenção de construções que resultam em uma tabela relacional e ligacional, respectivamente.

a) C <endereço de ponto> [EM <expressão de obtenção de construções>]

Este operador busca uma cópia da construção referenciada pelo <endereço de ponto>, que pode estar na base de dados, no canal auxiliar, ou que pode ter sido obtida por uma outra <expressão de obtenção de construções>.

exemplo:

C EMPREGADOS.TR

obtém uma cópia da tabela relacional do arquivo EMPREGADOS.

b) COLEC <endereço de ponto> [EM <obter construção>]

O resultado desta expressão é uma tabela relacional, ligacional ou uma coleção de itens, dependendo das construções colecionadas nos pontos endereçados. O termo <endereço de ponto> deve identificar tuplas, ligações ou itens (atributos de tuplas). Se o termo "EM <expressão de obtenção de construções>" for omitido, <endereço de ponto> é um endereço relativo ao contexto; do contrário, é relativo à construção obtida.

exemplos:

COLEC EMPREGADOS.TR.(C SALARIO > 20000)

esta expressão gera uma tabela relacional composta por todas as tuplas da tabela EMPREGADOS.TR onde o atributo SALARIO tem valor maior que 20.000.

COLEC EMPREGADOS.TR.(C SALARIO > 20000).NOMEEMP

a avaliação desta expressão gera uma coleção de itens composta pelos nomes dos funcionários cujos salários sejam maiores do que 20.000.

c) ESTREITAR <obter tabela relacional>{DE|PARA}<atributo> ,,,

Este operador produz uma tabela relacional pela eliminação de uma ou mais colunas da tabela obtida. A tabela resultante poderá apresentar cardinalidade menor que a de origem, quando for eliminado algum atributo componente da chave primária. Esta é a operação de projeção da álgebra relacional definida em [COD 70].

exemplo:

```
ESTREITAR C EMPREGADOS.TR PARA CODEMP,
                               DEPTOEMP,NOMEEMP
```

d) JUNTAR <obter tabela relacional> COM <obter tabela relacional>
POR C<atributo>{EXCL|INCL}=C <atributo>{EXCL|INCL} ,,,

Esta operação gera uma tabela relacional a partir de duas outras (extensão do "equi-join" da abordagem relacional [COD 70]). Será incluída uma tupla na tabela resultado cada vez que a condição de junção for satisfeita.

exemplo:

```
JUNTAR C EMPREGADOS.TR COM C DEPARTAMENTOS.TR
        POR  C DEPTOEMP = C CODDEPTO EXCL
```

e) <obter tabela relacional>{UNI|DIF|ISEC}<obter tabela relacional>

ou

<obter coleção de itens>{UNI|DIF|ISEC}<obter coleção de itens>

É obtida uma tabela relacional (primeiro caso) ou uma coleção de itens (segundo caso) pela união (UNI), diferença (DIF) ou interseção (ISEC) das duas construções obtidas. No primeiro caso, as duas tabelas relacionais obtidas devem possuir tuplas de mesma composição.

exemplo:

```
COLEC EMPREGADOS.TR.(C DEPTOEMP = 'D200').NOMEEMP
  ISEC
COLEC EMPREGADOS.TR.(C SALARIO < 30000).NOMEEMP
```

a coleção de itens resultante conterá o nome de todos os funcionários que trabalham no departamento D200 e ganham salário menor que 30.000.

f) LIGAR <obter tabela relacional> COM <obter tabela relacional>
POR C<atributo> = C <atributo> {EXCL|INCL} ,,,

O resultado desta operação é uma tabela ligacional que contém uma ligação para cada tupla da primeira tabela relacional obtida. O ligante é esta tupla e a tabela de ligados é constituída por todas as tuplas, da segunda tabela obtida, que satisfizerem a condição de ligação. Os atributos de ligação sempre fazem parte das tuplas de ligantes, podendo ou não pertencer às tabelas de ligados.

exemplo:

```
LIGAR C DEPARTAMENTOS.TR COM C EMPREGADOS.TR
  POR  C CODDEPTO = C DEPTOEMP INCL
```

o resultado deste exemplo é a tabela ligacional da figura 2.3.

g) AGRUPAR <obter tabela relacional> POR <atributo> ,,,

Esta expressão produz uma tabela ligacional a partir da relacional obtida. O processo de agrupar constitui-se da formação de conjuntos de tuplas que possuam os mesmos valores sob os atributos indicados. Cada um destes conjuntos dá origem a uma ligação onde a tupla do ligante é composta pelos atributos indicados e a tabela de ligados contém as tuplas do conjunto constituídas dos demais atribu-

tos.

exemplo:

AGRUPAR C EMPREGADOS.TR POR SALARIO

o resultado deste exemplo é uma tabela ligacional onde cada ligação abriga empregados que recebem um mesmo salário.

h) DESAGRUPAR <obter tabela ligacional>

Esta expressão tem efeito inverso ao da anterior. A partir da tabela ligacional obtida, é gerada uma tabela relacional, adicionando-se às tuplas de ligados os atributos de seus respectivos ligantes.

exemplo:

DESAGRUPAR (AGRUPAR C EMPREGADOS.TR POR SALARIO)

o resultado deste exemplo é uma tabela relacional igual a EMPREGADOS.TR.

i) COMPOR {TUP|LIG} (<nome de construção> :=
<obter{item|tupla|tabela relacional}>...)

Este operador produz uma tupla ou ligação a partir de construções da base de dados e/ou do canal auxiliar e/ou de variáveis Pascal do programa PLOBAN.

exemplos:

1) COMPOR TUP (CODEMP := 105
NOMEEMP := 'JOÃO CARDOSO'
SALARIO := 30000
DEPTOEMP := 'D100')

esta operação compõe uma tupla com a mesma composição de atributos das tuplas da tabela EMPREGADOS.TR.

```
2) COMPOR LIG (L:=COMPOR TUP(CODDEPTO:='D500'
                             NOMEDEPTO:='ALMOXARIFADO'
                             LOCAL:='SÃO PAULO')
               T:=COLEC AUX.NOVOSEMPREGADOS.(C DEPTOEMP =
                                              'D500')
              )
```

j) INT <variável Pascal>

Este operador internaliza para o sistema o conteúdo de uma variável Pascal do programa PLOBAN. Representa a operação de entrada de dados do sistema. Este operador é usado em associação com expressões de obtenção de construções do tipo COMPOR.

exemplo:

```
COMPOR TUP (CODEEMP:= INT <valor advindo da entrada 1>
            NOMEEMP:= INT <valor advindo da entrada 2>
            SALARIO:= INT <valor advindo da entrada 3>
            DEPTOEMP:= INT <valor advindo da entrada 4>)
```

esta operação compõe uma tupla a partir de valores de atributos contidos em variáveis Pascal do programa PLOBAN do usuário.

k) CARD <expressão de obtenção de construções>

Especifica um número real que indica a cardinalidade da construção. Esta construção só pode ser uma coleção de itens, uma tabela relacional ou uma tabela ligacional.

exemplo:

```
CARD (COLEC EMPREGADOS.TR.(C SALARIO>=35000).SALARIO)
```

o resultado deste exemplo será um valor real que expressa quantos valores de salário maiores ou iguais a 35.000 existem no arquivo de empregados (neste caso 2: 35000, 50000).

l) CONT <endereço de ponto> [EM <expressão de obtenção de construções>]

Produz um real obtido pela contagem dos pontos endereçados.

exemplo:

```
CONT EMPREGADOS.TR.(C SALARIO >= 35000).SALARIO
```

o resultado deste exemplo será um valor real que expressa o número de ocorrências de valores de salários maiores ou iguais a 35000 no arquivo de empregados (neste caso 3: 35000, 35000 e 50000).

m) {SOMA|MEDIA|MAX|MIN|DESV} SOBRE <endereço de ponto> [EM <expressão de obtenção de construções>]

O resultado desta expressão é um valor real que exprime a soma, a média, o máximo, o mínimo ou o desvio padrão do conjunto de pontos endereçados.

exemplo:

```
SOMA SOBRE COBTIDA.SALARIO
     EM COLEC EMPREGADOS.TR.(C DEPTOEMP='D200')
```

o resultado desta operação é o valor 55000, obtido pela soma dos salários dos funcionários que trabalham no departamento D200.

Comparando-se LOBAN com outras linguagens de operação de banco de dados como SQL [CHA 81] ou QUEL [STO 76] pode-se considerá-la como não-procedural. Apesar das instruções operativas e de controle de fluxo da linguagem imprimirem uma certa proceduralidade aos programas (a ordem de execução das instruções coincide com a ordem em que elas aparecem no programa ou é explicitamente definida pela ins-

trução de controle de fluxo) a execução dos operadores das expressões de obtenção de construções é, totalmente, não procedural. O usuário pode delegar ao sistema a tarefa de escolher a ordem e a melhor forma de avaliar as expressões.

Nas aplicações de consulta sem atualização, o usuário pode utilizar LOBAN de forma, francamente, não-procedural.

3. O SISTEMA L

3.1 Introdução

O Sistema L é o sistema de gerência de banco de dados, sendo desenvolvido no CPGCC da UFRGS, e que implementa um subconjunto significativo da linguagem LOBAN, apresentado na seção anterior.

A máquina L é, basicamente, uma máquina de pilha. Esta arquitetura foi escolhida [GOL 81] por ser a mais adequada a uma linguagem como LOBAN, onde existem variáveis declaradas explicitamente e aninhamento de expressões de obtenção de construções.

A grande diferença da máquina L para outras máquinas de pilha é a existência de operadores que trabalham, diretamente, sobre a base de dados interna, ao invés de constituirem-se em simples primitivas de entrada e saída.

3.2 Arquitetura anterior do Sistema L

A figura 3.1 apresenta, na forma de uma rede de estações e canais [RIC 77], o esquema de comunicação entre o usuário e o Sistema L [HEU 81]. Note-se que toda esta comunicação é feita através das instruções LOBAN submetidas, diretamente, ao sistema. Na figura, estão dispostas duas estações e cinco canais. Uma estação representa o usuário e a outra, o sistema propriamente dito. Os canais relacionados com o sistema são os de entrada e saída, interface LOBAN, base de dados e canal auxiliar.

O canal auxiliar mantém as estruturas criadas durante a execução do programa LOBAN (como, por exemplo, tabelas ligacionais) até o final desta.

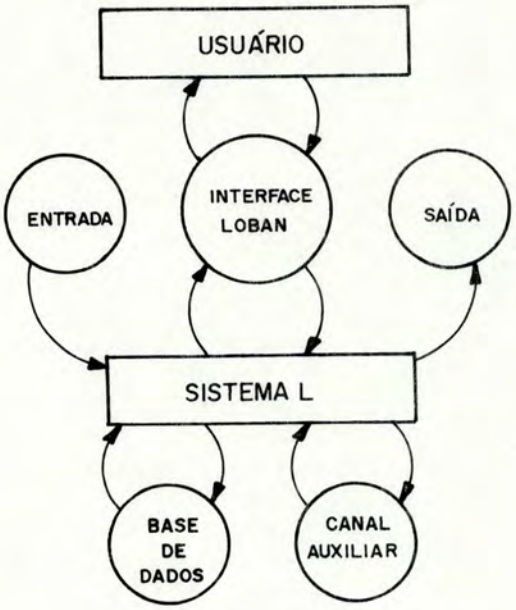

FIGURA 3.1: Comunicação do Usuário com o Sistema L

A figura 3.2 mostra a arquitetura anterior do Sistema L. Observa-se daí, duas estações e cinco canais. As estações são o Reconhecedor e o Interpretador de Pacotes.

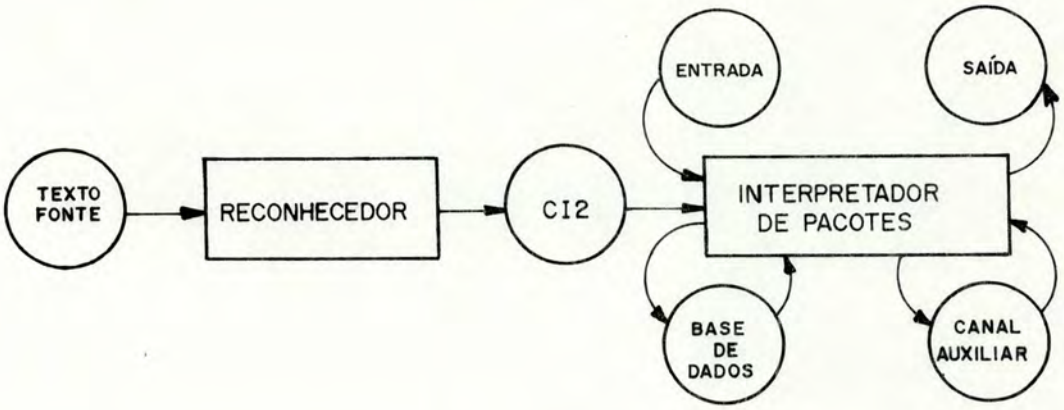

FIGURA 3.2: Arquitetura Antiga do Sistema L

No Reconhecedor é feito o processamento do texto fonte que independe de outros canais. A figura 3.3 apresenta a arquitetura do Reconhecedor que executa as análises léxica e sintática do texto fonte, transformando-o em um programa em código intermediário 2 (no canal, CI2).

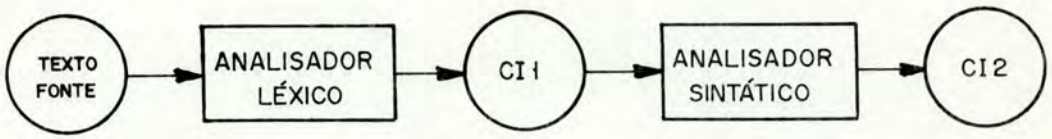

FIGURA 3.3: Arquitetura do Módulo Reconhecedor

Um programa em código intermediário 2 apresenta as seguintes características:

- é constituído de uma lista de unidades sintáticas, que contém referências a uma tabela de literais;

- está em notação polonesa, para facilitar o processamento pela próxima estação;

- está dividido em "pacotes", que são conjuntos de instruções que podem ser compilados e executados de uma só vez, pois nenhuma instrução do pacote (exceto a última) especifica alteração da estrutura dos canais.

A outra estação do Sistema L, denominada Interpretador de Pacotes, é encarregada de executar as instruções constantes nos pacotes à luz das estruturas de dados existentes nos canais.

A figura 3.4 mostra a arquitetura do Interpretador de Pacotes.

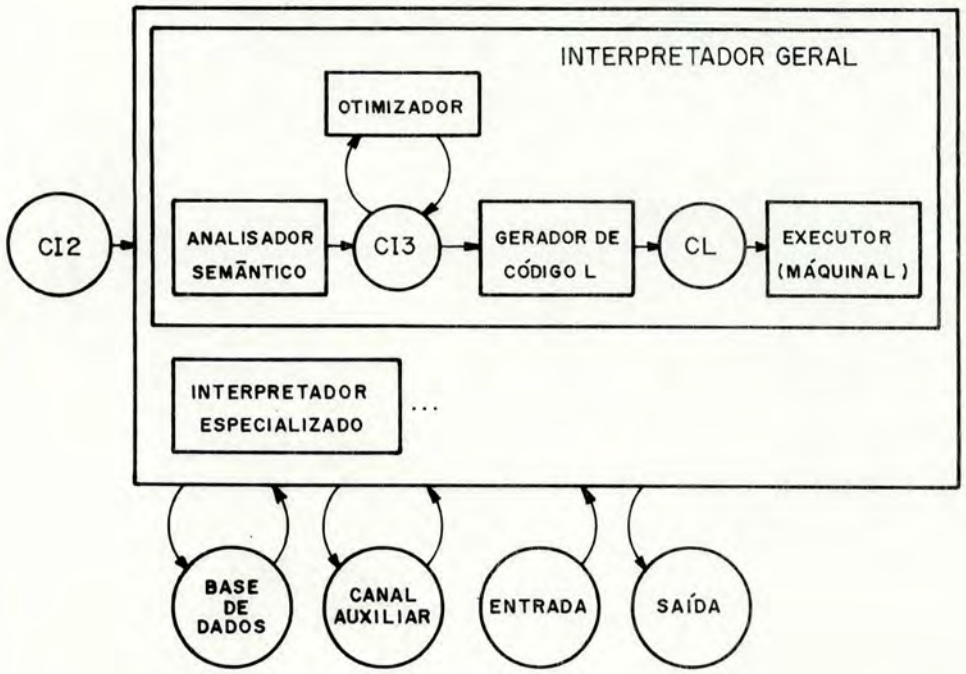

FIGURA 3.4: Arquitetura do Interpretador de Pacotes

Os interpretadores especializados têm por função executar as instruções administrativas de LOBAN, com exceção da instrução de controle de fluxo FAZER PARA CADA. Estes módulos do sistema trabalham na descrição da base de dados e em processos de reconstrução e cópia do banco de dados.

O interpretador geral executa os pacotes compostos por instruções operativas e de controle de fluxo. Esta estação contém quatro outras, cada uma desempenhando uma função distinta.

O Analisador Semântico verifica, contra a descrição da base de dados e canal auxiliar, a compatibilidade entre operandos e operadores. Além disso, traduz os nomes externos para identificadores internos do banco de dados. Este módulo gera o código intermediário 3 (CI3) onde cada instrução operativa é representada na forma de uma árvore.

O Otimizador lê o CI3 e tenta alterar a ordem dos operadores dentro das instruções, visando uma diminuição do volume de dados manipulado e conseqüente melhoria de performance durante sua execução. As técnicas utilizadas para este trabalho estão descritas em [GOL 80].

O Gerador de Código, usando como entrada o código intermediário 3, tem por função gerar o código L, que será interpretado pelo módulo Executor. Esta última estação é a própria Máquina L que resolve, em última análise, as instruções operativas LOBAN, submetidas pelo usuário.

4. O AMBIENTE DE IMPLEMENTAÇÃO

O Sistema L, como um todo, está sendo implementado em um minicomputador LABO 8034, com 96 Kbytes de memória principal e 20 Mbytes de memória secundária em disco magnético (duas unidades, de 5 Mbytes cada, de discos removíveis e duas unidades, de 5 Mbytes cada, de discos fixos).

Sobre este computador, foi implementado o Sistema Pascal Concorrente [MED 81] apresentado em [BRI 77] e,a partir dele, foi construído o Sistema Operacional para LOBAN (SOL). Trata-se de um sistema desenvolvido para atender as necessidades específicas do Sistema L. O SOL, descrito em [CAR 82], além de outras funções, implementa o método de acesso à base de dados e ao canal auxiliar.

Escrito em Pascal Concorrente, o SOL está baseado na filosofia de processos envelopes onde, em um ambiente de multiprogramação, cada processo gerencia e é responsável pela execução de um programa seqüencial (escrito em Pascal Seqüencial).

Três dos processos do SOL são gerentes de arquivos (SERPPROCESS, RELPROCESS e INVPROCESS) e, em conjunto, implementam o método de acesso oferecido pelo Sistema. O Sistema L é executado por um outro processo (JOBPROCESS) que se comunica com os anteriores para manipular a base de dados e o canal auxiliar.

O SOL oferece quatro organizações de arquivos distintas: serial, relacional, de inversão, de associação.

O Sistema L manipula estes arquivos a partir de primitivas de acesso passadas aos gerentes de arquivo. Existem primitivas administrativas de uso geral e outras específicas de cada organização. A seguir, são apresentadas aquelas primitivas mais importantes para este trabalho.

- *Primitivas de gerência de arquivos:*

a) OPEN: torna disponível, para o programa, o arquivo especificado como parâmetro;

b) CLOSE: materializa as alterações feitas no descritor do arquivo, pelo programa, tornando-o não disponível ao usuário.

- *Primitivas de acesso a arquivos seriais:*

Os arquivos seriais são suportados pelo SOL para uso exclusivo do Sistema (programas fonte e objeto, arquivos temporários, etc.).

a) READ: esta primitiva transporta, para uma área do programa, um registro de determinado arquivo serial;

b) WRITE: provoca a transferência de um registro, em área de programa, para o arquivo serial especificado.

- *Primitivas de acesso a arquivos relacionais:*

a) INCDATA: transfere uma tupla da área do usuário para a próxima posição disponível de um arquivo especificado;

b) EXCDATA: libera a posição do arquivo, que contém a tupla especificada; os dados constantes nesta tupla, tornam-se assim inacessíveis;

c) ALTDATA: a tupla contida no endereço indicado é substituída por outra da área do programa;

d) GETDATA: copia o conteúdo da tupla, de endereço especificado, na área do programa, também especificada;

e) SEQDATA: copia o conteúdo da próxima tupla do arquivo em uma área especificada.

- *Primitivas de acesso a arquivos de inversão:*

a) INCINV: esta primitiva estabelece a ligação entre uma chave do arquivo de inversões e uma tupla de arquivo relacional ao qual está associado;

b) EXCINV: desfaz a ligação entre uma tupla de arquivo relacional e um registro do arquivo de inversões;

c) SEQINV: este procedimento permite a recuperação de registros de inversão, ordenados pela chave de inversão, em ordem crescente ou decrescente do valor da chave;

d) GETINV: esta primitiva promove a recuperação de um registro do arquivo de inversões, do qual é conhecida a chave de inversão.

- *Primitivas para administração da memória:*

Estas primitivas controlam a alocação de espaço no HEAP (área de variáveis dinâmicas do programa seqüencial).

a) MARK: marca em uma variável do sistema (TOP) o próximo endereço (byte) do HEAP a ser utilizado;

b) RELEASE: o apontador de próximo endereço do HEAP assume o valor da variável TOP do sistema.

- *Primitiva para chamadas a rotinas externas:*

RUN: este procedimento permite transferir o controle de um programa seqüencial a outro. Ao final da execução do programa chamado, o controle volta ao chamador, no próximo comando deste.

Muitas decisões tomadas no projeto do gerador de código levaram em consideração limitações apresentadas por este ambiente de implementação. Estas limitações serão abordadas nas próximas seções.

5. JUSTIFICATIVAS PARA A PROPOSTA DE UMA NOVA ARQUITETURA PARA O SISTEMA L

Diversos motivos contribuíram para a proposta de uma nova arquitetura do Sistema L. Estes motivos foram sendo conhecidos a partir da experiência adquirida no esforço de implementação do protótipo e das experiências de outros grupos na construção de sistemas semelhantes (exemplo: Sistema INGRES [STO 80] e Sistema R [CHA 81]).

Observou-se que, com o esquema interpretativo da arquitetura anterior, todo o esforço de reconhecimento, análise semântica e otimização dos pacotes seria perdido após a execução dos mesmos, ou seja, a re-execução de um mesmo programa LOBAN exige que este seja, novamente, submetido a todos os módulos do sistema. Isto representa uma degradação significativa da performance do sistema, na execução repetitiva de consultas do usuário.

Outro fato observado, e que reforçou a idéia de uma mudança na arquitetura antiga, é o do código L ser muito semelhante ao código P gerado pelo compilador Pascal (exceção feita às primitivas de acesso à base de dados). Portanto, substituindo-se o código L pelo código P,poder-se-ia suprimir a implementação da máquina L poupando-se, assim, esforços na etapa de desenvolvimento do sistema. Esta modificação ainda aumentaria a performance do código gerado, em tempo de execução já que, na arquitetura antiga, este seria interpretado pela máquina L que, por sua vez, seria interpretada pela máquina P.

O estudo da utilização do compilador Pascal, como ferramenta de auxílio na compilação de programas LOBAN, despertou os implementadores do sistema para a possibilidade de embutir a linguagem LOBAN na linguagem Pascal, provendo assim o usuário da flexibilidade obtenível através da combinação de uma linguagem voltada à manipulação de bancos de dados com outra de propósitos gerais. Outras vantagens que

poderiam advir de um [illegible] as reduções de tempo e esforço na fase de implementação do sistema, pois as funções de entrada e saída e a resolução de expressões poderiam ser feitas, pelo usuário, na linguagem hospedeira.

A partir destas considerações e de estudos de viabilidade feitos sobre elas, decidiu-se alterar a arquitetura do Sistema L tornando-o compilativo e oferecendo, ao usuário, a possibilidade de programas LOBAN embutidos em Pascal. Estabeleceu-se, porém, que a nova arquitetura deveria ser projetada de maneira tal que o trabalho sendo desenvolvido, até o momento, para a implementação do sistema, não fosse perdido.

Partiu-se, então, para um estudo mais aprofundado das arquiteturas e esquemas compilativos de outros sistemas de gerência de banco de dados, baseados na abordagem relacional e que suportam linguagens de operação não-procedurais. Estes estudos são apresentados no próximo capítulo. Além disso, pesquisou-se técnicas de geração de código em outros sistemas como o REPORTER e o DMS II, ambos desenvolvidos pela Burroughs.

De posse das conclusões tiradas a partir destes estudos e conhecendo-se as características próprias do LOBAN e do ambiente de sua implementação, propôs-se uma nova arquitetura e um esquema compilativo para o Sistema L.

6. COMPILAÇÃO DE LINGUAGENS NÃO-PROCEDURAIS

6.1 Introdução

A característica principal das linguagens não-procedurais é permitir ao usuário manipular seus dados sem ter conhecimento de sua estrutura interna (independência de dados) [DAT 77]. O usuário indica, através das instruções, "o que" manipular sem precisar definir "como" esta manipulação será executada. O sistema que suporta a linguagem é que deve conhecer a estrutura interna dos dados para projetar estratégias de acesso adequadas a eles. Também é de responsabilidade do sistema a otimização destas estratégias.

Exemplo de linguagens de operação de banco de dados não-procedurais são aquelas baseadas em álgebra ou cálculo relacionais, como SQL [IBM 81], QUEL [STO 76] e LOBAN [SAN 80].

A compilação de linguagens não-procedurais apresenta uma série de diferenças em relação à compilação de linguagens de alto nível do tipo procedural. Algumas das razões para estas diferenças, descritas em [PRY 83], são apresentadas abaixo:

a) a ordem em que as operações aparecem nas instruções pode ser irrelevante. O sistema é que deve analisar problemas de dependência entre variáveis e instruções dando, assim, a ordem de execução das mesmas;

b) as instruções referenciam os dados sem determinar se estes estão na memória ou em periféricos. É necessário o sistema verificar a descrição dos dados e a ordem das instruções, para determinar as operações de I/O;

c) várias instruções operam com conjuntos inteiros de dados (ex.: tabelas relacionais e ligacionais em LOBAN) sendo necessário, na sua compilação, implementar la-

ços ("loops") de processamento;

d) a tarefa de mapear os objetos de dados, descritos pelo usuário, para a memória também é função do sistema. Este deve projetar o uso da memória principal de maneira eficiente.

Todas as características apresentadas acima para ilustrar a compilação de linguagens não-procedurais podem ser encontradas nas expressões de obtenção de construções de LOBAN.

6.2 Compilação de linguagens não-procedurais em sistemas existentes

6.2.1 Os diferentes esquemas compilativos

A maioria das linguagens não-procedurais de operação de banco de dados são usadas como interface de sistemas de gerência de banco de dados calcados sobre a abordagem relacional. Na verdade, a estrutura destas linguagens é fruto das facilidades oferecidas ao usuário por esta abordagem. Algumas destas facilidades são:

a) o usuário tem uma visão simplificada de seus dados (na forma de tabelas);

b) o usuário não precisa conhecer a estrutura interna da base de dados (independência de dados).

Sempre existiu um certo pessimismo em relação à performance oferecida pela abordagem relacional. A razão deste sentimento deve-se ao fato do usuário não poder navegar, explicitamente, pelos caminhos de acesso. Além disso, havia, no início, uma forte tendência de implementar sistemas de gerência de banco de dados que interpretam as consultas do usuário (ex.: Sistema INGRES [ALL 76]).

Surgiu, então, a idéia de compilar as linguagens de consulta, aumentando assim a performance dos sistemas, pela passagem de parte do processamento das instruções não-procedurais para a fase de compilação, diminuindo o tempo para a execução propriamente dita [LOR 79].

Como deve-se notar, mais adiante, tanto os esquemas compilativos como os interpretativos apresentam vantagens e desvantagens em relação à performance dos sistemas. Atualmente, já estão surgindo propostas de esquemas híbridos [KAT 79].

As principais linguagens de operação de sistemas relacionais são oferecidas ao usuário embutidas em linguagens procedurais de propósito geral. Estas linguagens são oferecidas também para consultas de forma interativa. Exemplos disso são o SQL do Sistema R, embutido em PL/I ou COBOL [CHA 76] e o EQUEL do Sistema INGRES que tem, como hospedeira, a linguagem de programação C [ALL 76]. A vida útil de programas escritos nestas linguagens pode ser dividida em três tempos distintos [KAT 79], como segue:

a) tempo de pré-processamento (ou pré-compilação): neste tempo o sistema identifica e prepara as instruções não-procedurais do programa;

b) tempo de compilação: o programa é submetido ao compilador da linguagem hospedeira que gera, a partir daí, código executável e chamadas ao sistema de gerência de banco de dados;

c) tempo de execução: o código gerado pelo compilador é executado e as consultas, feitas pelo usuário ao banco de dados, são processadas.

Existem várias etapas que devem ser cumpridas a fim de processar as consultas do usuário, em linguagem não-procedural:

1) reconhecimento léxico e sintático ("parsing");

2) tradução de nomes externos em nomes internos (através de um catálogo) e análise semântica das consultas;

3) construção de um plano de processamento (seleção de caminhos de acesso);

4) execução do plano de processamento com chamadas ao método de acesso.

Variando-se o tempo em que as etapas acima são processadas pelo sistema, pode-se definir cinco esquemas compilativos [KAT 79]:

Ø - a consulta é completamente interpretada, em tempo de execução (etapas 1 a 4);

1 - as análises léxica e sintática das consultas são feitas em tempo de pré-processamento, traduzindo-as para alguma forma interna (ex.: "parse tree"), que depois será interpretada;

2 - o pré-processador trabalha a consulta até a análise semântica, com a tradução completa dos nomes, se possível;

3 - a seleção de caminhos de acesso também é feita em tempo de pré-processamento, sendo seu resultado mantido em uma forma interna, para ser interpretado em tempo de execução;

4 - a seleção de caminhos de acesso é feita, como no esquema 3 porém seu resultado é representado, diretamente, no programa do usuário como chamadas ("calls") ao método de acesso. A consulta é totalmente compilada em tempo de pré-processamento.

Algumas características dos sistemas mais interpretativos (esquemas 0, 1 e 2) são:

a) o tempo de execução da consulta é, geralmente maior, pois esta é interpretada por outro programa;

b) em tempo de execução, o espaço necessário de memória é grande, já que deve-se somar ao tamanho da consulta, o tamanho do programa interpretador;

c) o interpretador pode ser escrito em linguagem de alto nível facilitando, em muito, o desenvolvimento, a manutenção e a documentação do sistema além de deixá-lo menos sensível a modificações de hadware e software;

d) em tempo de execução dispõe-se de informações mais precisas, para auxiliar na decomposição das consultas (ex.: cardinalidade das relações);

e) a seleção de caminhos de acesso, em tempo de execução, também pode ser mais precisa, pois o sistema tem uma visão atualizada do banco de dados no que se refere às estruturas de acesso e informações administrativas (ex.: autorizações e "views").

As principais características dos esquemas mais compilativos (esquemas 3 e 4) são descritas abaixo:

a) o tempo de execução das consultas pode diminuir, consideravelmente, já que grande parte do processamento destas é feito em tempos anteriores e o código gerado é executado diretamente pela máquina;

b) mais espaço de memória pode ser reservado aos dados manipulados pelas instruções, graças à inexistência do programa interpretador;

c) o sistema deve basear-se em estimativas para proceder a decomposição das consultas e a seleção de caminhos de acesso, já que a situação do banco de dados em tempo de pré-processamento pode não ser a mesma do tempo de execução. Isto pode degradar a performance do sistema neste

último tempo;

d) em sistemas que seguem o esquema compilativo 4 (ex.: Sistema R [LOR 79]) é necessário criar mecanismos que possibilitem a recompilação de consultas em tempo de execução, caso venham a acontecer modificações no banco de dados que assim o exijam (ex.: eliminação de um índice que era utilizado);

e) a compilação é tanto mais eficaz quanto mais estável for o banco de dados. Além disso, este esquema é mais eficiente em sistemas que utilizam linguagens embutidas (utilizadas por programadores mais experientes) do que em ambientes voltados a usuários não treinados, para consultas "ad hoc" [KAT 79].

6.2.2 Alguns Sistemas e seus Esquemas de Compilação

a) ASTRA [RIS 77]

Sistema relacional construído sobre um hierárquico na Universidade de Trondheim, Noruega. Suporta a linguagem ASTRAL (A Structured Relational Applications Language) baseada em SIMULA.

Em tempo de pré-processamento, a consulta em ASTRAL é transformada em rotinas em SIMULA com chamadas ao método de acesso. As chamadas são interpretadas em tempo de execução (esquema 3 de compilação).

b) PL/I Estendido [SUM 75]

Trata-se da linguagem PL/I com facilidades para a definição de tabelas e comandos do tipo "SELECT". Segue o esquema 4 de compilação. Os autores do projeto acreditam que a melhora na performance através do esquema compilativo pode ser minimizada (e até eliminada) pela sensibilidade do

programa compilado a mudanças no esquema.

c) Sistema INGRES [STO 76]

O Sistema INGRES, desenvolvido na University of California at Berkeley, suporta duas interfaces com o usuário [STO 76]. O QUEL (Query Language), interface autônoma, totalmente interpretada e o EQUEL (Embedded QUEL), interface embutida na linguagem de programação C. Os programas em EQUEL passam por um pré-processador, cuja única tarefa é transformar as instruções QUEL, do programa em C, em parâmetros de chamadas, que serão feitas em tempo de execução, ao interpretador do Sistema. Portanto o Sistema INGRES suporta o esquema compilativo 0. Todo o processamento das consultas é feito em tempo de execução.

d) Data Base Machine DIRECT [BOR 82]

Este sistema implementa o Sistema INGRES em um multiprocessador dedicado e especialista. Apesar de o Sistema INGRES em sua versão standard suportar um esquema interpretativo para processar as consultas em EQUEL, os projetistas do DIRECT implementaram um esquema compilativo semelhante ao 4, descrito acima. Para eles, a facilidade de construir e manter um sistema interpretativo não compensa sua baixa performance.

e) Sistema R [CHA 81]

O Sistema R foi um projeto experimental da IBM, na área de sistemas de gerência baseados na abordagem relacional. Desenvolvido no Laboratório de Pesquisas de San José da Califórnia, desde 1975, foi dado como encerrado em 1978. A partir de 1979, começou a ser comercializado, pela IBM, sob o nome SQL/DS.

Este foi um dos primeiros projetos de sistemas relacionais a pensar uma alternativa para o esquema interpre-

tativo. Seu esquema é o de nível 4. O programa hospedeiro faz chamadas aos módulos de acesso, totalmente compilados, que resolvem as consultas ao banco de dados. Além do código executável, os módulos de acesso mantém, cada um, o resultado da fase de reconhecimento da consulta do usuário ("parse tree") para agilizar a recompilação, caso esta se faça necessária.

É importante observar que este esquema compilativo também é aplicado nas consultas "ad hoc" do usuário.

6.2.3 Geração de Código em Sistemas Compilativos

6.2.3.1 Introdução

Nesta seção, apresenta-se os métodos de geração de código adotados por alguns dos sistemas de gerência de banco de dados relacionais que seguem os esquemas compilativos 3 e 4 apresentados na subseção 6.2.1.

6.2.3.2 Geração de código no Sistema R [LOR 79]

O sistema trabalha com a linguagem intermediária ASL (Access Specification Language), para a qual é traduzida a consulta do usuário após a seleção de caminhos de acesso. Em ASL, a estratégia de acesso é descrita de maneira procedural.

O pré-processador dispõe de cerca de 30 "modelos" ASL. Cada "modelo" é um algoritmo em ASL que resolve um problema elementar bem específico. Qualquer consulta do usuário, por mais complexa que seja, pode ser traduzida para uma combinação destes modelos.

A combinação de "modelos" ASL, para resolver uma

consulta, forma uma estrutura em árvore que é passada para o gerador de código. Cada "modelo", por sua vez,corresponde a um conjunto de "fragmentos" que são trechos,em assembler, compilados e link-editados ao pré-processador. Existem cerca de 100 "fragmentos" cuja combinação representa todos os "modelos" ASL.

A tarefa do gerador de código é, a partir da varredura da árvore de "modelos" ASL, produzir o módulo de acesso que resolve a consulta do usuário, a partir da combinação dos "fragmentos" de código.

Foi tomada a decisão de gerar-se, diretamente, código de máquina, a partir das consultas, ao invés de rotinas em linguagem hospedeira (PL/I ou COBOL), pois assim evita-se de submeter a consulta também ao compilador da linguagem hospedeira. Com isto, melhora-se a performance do sistema também em tempo de compilação.

Os algoritmos escritos em assembler podem ser mais eficientes, em tempo de execução, em comparação aos algoritmos escritos em linguagem de alto nível.

As consultas "on line" em SQL (interface autônoma do Sistema R) também são compiladas da mesma forma que as embutidas em PL/I ou COBOL. A diferença entre elas é que as primeiras são imediatamente executadas após a compilação. Esta decisão simplificou a arquitetura do Sistema e diminuiu seu tempo de desenvolvimento [CHA 81].

Através de testes ficou comprovado que a melhora de performance em tempo de execução devida à compilação das consultas, mesmo no caso de consultas não complexas, supera o "overhead" causado pelo esforço maior em tempo de pré-processamento [LOR 79],[CHA 81].

Algumas desvantagens, apresentadas por este método, podem ser apontadas:

a) a dificuldade de documentação do gerador pelo fato de os segmentos, todos, estarem escritos em assembler;

b) dificuldade de expansão do conjunto de instruções pois, para modificar os fragmentos ou aumentar seu número é necessário recompilá-los e link-editá-los ao pré-processador.

6.2.3.3 Geração de código no Sistema INGRES

O Sistema INGRES, como existe atualmente, segue um esquema interpretativo na execução de consultas do usuário.

Recentes estudos da performance do sistema, apresentados em [HAW 79] mostram que seria vantajoso executar as etapas de reconhecimento ("parsing") e validação (análise semântica) em tempo de pré-processamento, pelo menos para consultas simples (que só manipulam uma relação).Em transações mais longas, contudo, as atividades arroladas acima têm pouco impacto no tempo total da consulta devido ao significativo aumento no tempo de entrada e saída para a recuperação dos dados, em disco [STO 83].

A partir destes estudos, foi proposto um esquema híbrido de geração de código para aumentar a performance do sistema.

A solução alternativa, baseada num esquema compilativo, e que não exigisse uma reestruturação profunda do sistema, seria o que se convencionou chamar de compilação dinâmica.

O pré-processador compila somente consultas simples (que se referem a uma única relação) e partes de consultas complexas que serão repetidas muitas vezes, em tempo de execução (ex.: conversão de formatos).

Segundo os autores desta estratégia, sua vantagem em relação aos sistemas puramente compilativos é que a seleção de caminhos de acesso continua sendo feita em tempo de execução. Devido a complexidade desta tarefa, ela pode ser agilizada por um interpretador. Além disso a seleção de caminhos de acesso é mais eficiente, quando feita em tempo de execução.

O Dynamically Compiled INGRES (DC-INGRES) como é chamado este esquema de compilação, produz um tipo de código, de alguma forma, baseado no conceito de "threaded-code" [BEL 73]. Em comparação com a geração de código tradicional, o "overhead" do DC-INGRES, por ser baseado em "threaded-code", é da ordem de 10 a 20 por cento.

6.2.3.4 Geração de código na máquina de banco de dados DIRECT [BOR 82]

A máquina DIRECT é um sistema multi-microprocessado construído, especificamente, para executar o sistema relacional INGRES.

O esquema de compilação escolhido por seus projetistas é o de nível 4 apresentado na seção 6.2.1.

O pré-processador, a partir da análise da consulta, identifica as operações relacionais básicas a serem realizadas, bem como as informações próprias de cada consulta para realizá-las (ex.: identificação da operação "Join" e das relações nela envolvidas). A partir daí são selecionadas partes invariáveis já compiladas das rotinas, que são concatenadas com as partes de comparação específicas de cada consulta. O conjunto destas partes forma o código objeto completo da consulta.

Uma das características deste esquema é o aumento de performance na compilação da consulta pela utilização de

trechos genéricos já compilados. Estes mesmos trechos podem ser recombinados na compilação de outras consultas.

Um exemplo para ilustrar o conceito de trechos genéricos pode ser dado a partir do algoritmo de execução do operador "RESTRICT" da álgebra relacional que corresponde ao "ESTREITAR" de LOBAN. Os trechos de abertura e fechamento do arquivo e o laço de recuperação de registros podem ser usados no algoritmo independentemente do arquivo que será manipulado ou da expressão que selecionará as tuplas que deverão ser processadas.

7. NOVA ARQUITETURA PROPOSTA PARA O SISTEMA L

7.1 Introdução

A nova arquitetura proposta para o Sistema L apresenta duas diferenças básicas em relação à arquitetura antiga:

- o sistema dispõe agora de duas interfaces para a comunicação com o usuário. Uma interface para a descrição da base de dados e outra para a manipulação dos dados nela contidos. A figura 7.1 apresenta este novo esquema de comunicação entre o usuário e o sistema;

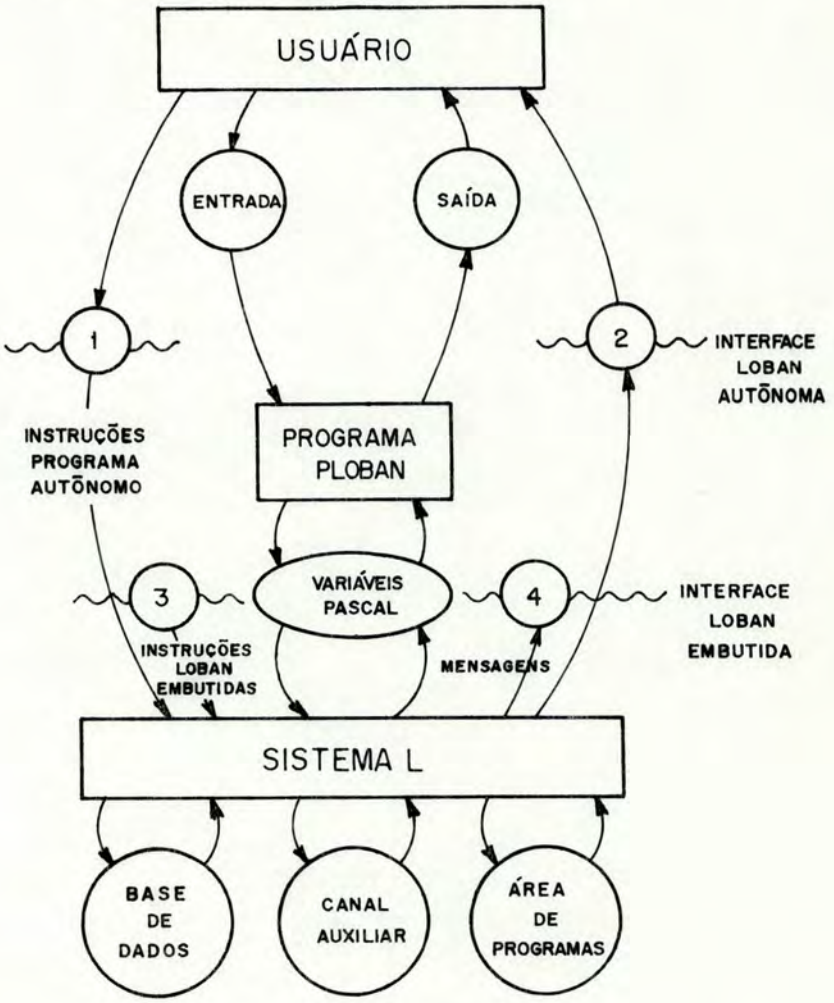

FIGURA 7.1: Novo Esquema de Comunicação entre Usuário e o Sistema L

- a interface de manipulação da base de dados segue um esquema compilativo que gera código P ao invés de código L e que processa o programa do usuário em duas etapas: pré-processamento e execução. Durante o pré-processamento é que se dá a compilação do programa.

Estas alterações de projeto visam diminuir o tempo de execução das instruções de manipulação do usuário, através da transferência de parte do processamento destas para a etapa de pré-processamento (que existirá somente uma vez para cada consulta) e, pela geração de código P permitindo, assim, que as instruções sejam executadas diretamente pela máquina P.

A nova arquitetura também diminui o esforço de implementação do sistema já que todo o tratamento de entrada e saída será implementado pelas instruções da linguagem hospedeira, no caso o Pascal.

Na figura 7.1 aparecem três estações que representam o usuário do sistema, o sistema de gerência e o programa PLOBAN (programa Pascal com instruções LOBAN embutidas).

O novo esquema, como já foi dito, dispõe de duas interfaces. A primeira, *Interface LOBAN Autônoma*, projetada para ser utilizada por administradores de banco de dados, permite a definição dos dados, incluindo a definição de restrições de integridade como chaves primária e estrangeira, e de aspectos de controle de acesso e reconstrução da base de dados.

Basicamente, a Interface LOBAN Autônoma é implementada pelos interpretadores especializados previstos na arquitetura antiga do sistema.

A segunda interface do sistema, chamada *Interface LOBAN Embutida*, permite ao usuário, a manipulação da base de dados através das instruções operativas LOBAN. A comunicação de dados entre o sistema de gerência e o programa do

usuário é feita através de variáveis Pascal do programa PLOBAN utilizadas nas instruções LOBAN embutidas. A comunicação do programa PLOBAN com o meio externo é implementada pelas instruções Pascal de entrada e saída. Além das instruções operativas, a interface embutida oferece a instrução LOBAN de controle de fluxo (FAZER PARA CADA).

O esquema proposto apresenta, ainda, três outros canais a saber:

- canal base de dados (contém todas as estruturas definidas pelo administrador da base de dados através da interface autônoma);

- canal auxiliar que suporta as estruturas de dados criadas dentro do programa PLOBAN. Estas informações ficam inacessíveis, ao usuário, após o fim da execução do programa;

- área de programas, onde ficam armazenados os programas PLOBAN e seus códigos objeto além dos programas do sistema.

A única modificação feita sobre a linguagem LOBAN para melhor adaptá-la à nova arquitetura, foi nos tipos de dados oferecidos para a definição de atributos de tuplas. Ao invés dos tipos de dados originais da linguagem(NUMCAR1, NDEC1, IDEC1, RDEC1, REAL1), LOBAN suporta, agora, os tipos elementares do Pascal (INTEIRO, REAL, CARACTER e conjunto de caracteres). Esta decisão foi tomada para tornar mais homogênea a interface entre os ambientes de LOBAN e Pascal no programa PLOBAN, evitando assim, o trabalho oneroso de conversão de formatos [STO 77].

7.2 O Programa PLOBAN

7.2.1 A Estrutura do Programa PLOBAN

O programa PLOBAN é um programa, escrito em Pascal, com instruções LOBAN embutidas. Ele pode ser subdividido em três partes que devem obedecer a seguinte seqüência:

a) bloco inicial: indica qual o acervo setorial (banco de dados LOBAN) que será manipulado pelo programa, fornecendo também, o termo de apresentação do usuário. É neste bloco que devem ser especificados os tipos de construções possíveis no canal auxiliar, caso o usuário pretenda utilizá-lo em tempo de execução. O bloco inicial é composto, exclusivamente, por instruções LOBAN;

b) bloco de instruções: representa o programa Pascal propriamente dito, contendo instruções LOBAN operativas e de controle de fluxo para a manipulação da base de dados e canal auxiliar;

c) bloco de encerramento: é composto, unicamente, pela instrução LOBAN ENCERRAR que finaliza o programa PLOBAN.

As linhas do programa PLOBAN que contêm instruções LOBAN devem ser iniciadas com o caractere ## na primeira coluna. Cada linha pode conter mais de uma instrução LOBAN e, uma mesma instrução pode estender-se por mais de uma linha. A única restrição imposta é que instruções Pascal não podem ser dispostas, em uma mesma linha de código, com instruções LOBAN.

7.2.2 Exemplo de Programa PLOBAN

O programa apresentado, a seguir, faz com que sejam impressos os nomes de todos os empregados que trabalham na cidade de Porto Alegre (ver figuras 2.1 e 2.3).

```
1  ##  USAR ACSET EMPREGADOS-DEPARTAMENTOS-DIVISÕES
2  ##       PARA USUARIO OLIVEIRA
   :
3  TYPE     TYPENOME = ARRAY [1..30] OF CHAR;
   :
4  VAR      EMPNOME : TYPENOME;
   :
5  PROCEDURE  IMPRIMENOME (NOME : TYPENOME);
                    :
   :
6  ##   FAZER PARA CADA TUPLA
7  ##        EM (COLEC (C L.LOCAL='PORTO ALEGRE').T.TUP
8  ##              EM (LIGAR DEPARTAMENTOS.TR COM EMPREGADOS.TR
9  ##                    POR C CODDEPTO = C DEPTOEMP INCL))
10 ##   ((
11 ##        REPRESENTAR (C PC.NOMEEMP EM EMPNOME)
12           IMPRIMENOME (EMPNOME);
13 ##   ))
   :
14 ##   ENCERRAR
```

O bloco inicial do programa acima aparece nas linhas 1 e 2. Já as linhas numeradas de 3 a 13 representam o bloco de instruções, enquanto a linha 14 compõe o bloco de encerramento do programa PLOBAN.

As linhas de 10 a 13 representam o bloco de instruções que será executado, uma vez, para cada tupla obtida pelo operador COLEC que, por sua vez, recebe ligações do operador LIGAR (linhas 7 a 9). A linha 12 contém uma instrução Pascal. Trata-se de uma chamada à rotina IMPRIMENOME.

7.3 Apresentação da nova arquitetura

A figura 7.2 apresenta a arquitetura proposta para o Sistema L. Nela, o texto fonte (que pode ser um programa LOBAN ou PLOBAN) é submetido a um módulo reconhecedor comum às duas interfaces. Este módulo é, basicamente, o mesmo apresentado na figura 3.3 como parte da arquitetura anterior. Sua única diferença em relação àquele, é a capacidade de identificar as instruções LOBAN do programa LOBAN ou PLOBAN, só então realizando as análises léxica e sintática. O reconhecedor produz, como resultado de seu processamento, o código intermediário 2 representado, na figura, pelo canal CI2. No caso do texto fonte ser um programa PLOBAN, o reconhecedor também grava, no CI2, marcas que indicam a localização de trechos Pascal do programa do usuário. Estas informações são importantes para o módulo gerador de código, como poderá ser observado mais tarde.

Se o texto fonte é um programa LOBAN, o CI2 é processado por um interpretador especializado que executa, automaticamente, o programa.

Caso o texto fonte representar um programa PLOBAN, o CI2 é submetido a um pré-processador que produz código P executável mais uma lista com os nomes das estruturas de dados que serão manipuladas, pelo programa, em tempo de execução. Estes dados, juntamente com o CI2, são armazenados no canal programa objeto (CPO).

O pré-processador não executa o programa PLOBAN, automaticamente. Para executá-lo, o usuário deve acionar a estação Executor PLOBAN. Este módulo verifica se as estruturas de dados que o programa manipulará continuam válidas. Em caso afirmativo, o sistema passa a executar o código P gerado pelo pré-processador. Se não, o executor submete o CI2, mantido no canal programa objeto, novamente ao pré-processador para que este recompile o programa PLOBAN.

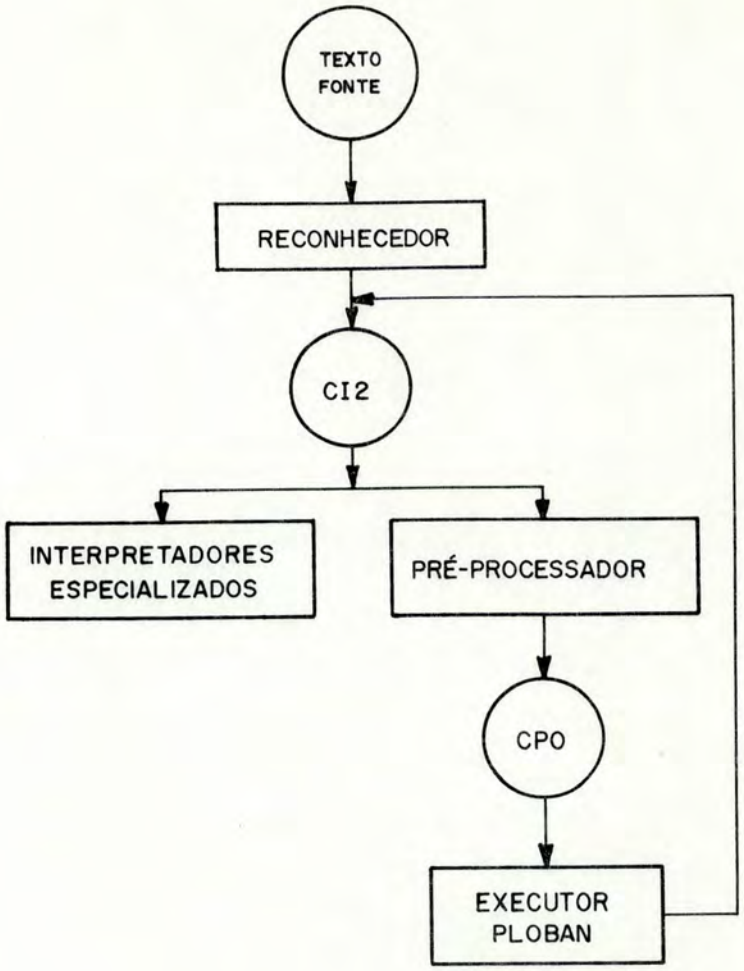

FIGURA 7.2: Nova Arquitetura Proposta para o Sistema L

A principal vantagem de processar o programa PLOBAN em duas etapas (pré-processamento e execução) é que o usuário pode executar o código gerado tantas vezes quantas desejar.

O fato de manter-se o CI2 junto com o código objeto do programa PLOBAN agiliza sua recompilação, já que torna-se desnecessário submetê-lo novamente ao reconhecedor.

7.4 Compilação de Programas PLOBAN

7.4.1 Introdução

A partir das alternativas para a execução de consultas a banco de dados apresentadas em [KAT 79] e, do estudo de soluções a este problema adotadas em sistemas já existentes, apresenta-se aqui, uma proposta para a execução de consultas através do programa PLOBAN.

A escolha de um esquema interpretativo ou parcialmente interpretativo poderia degradar sensivelmente a performance do sistema pelos motivos apresentados na capítulo 6. Além disso, a própria máquina Pascal, sobre a qual se está construindo o sistema L, utiliza-se de um esquema interpretativo, o que já imprime um ritmo relativamente lento à execução dos programas a ela submetidos.

Decidiu-se então por um esquema compilativo tentando-se antecipar, para o tempo de pré-processamento, o maior número possível de tarefas necessárias ao processamento da consulta, aumentando assim, a eficiência do sistema ao tempo de execução do programa PLOBAN.

O esquema compilativo proposto corresponde ao de número 4 apresentado na seção 6.2.1, que compila toda a consulta em tempo de pré-processamento. O programa PLOBAN passa por três etapas:

O pré-processador do sistema transforma as consultas LOBAN do programa em rotinas escritas em Pascal. O programa Pascal resultante é, então, compilado e, o resultado desta compilação é um programa objeto em código P executável.

Os possíveis problemas causados pela realização da seleção de caminhos de acesso, em tempo de pré-processamento, podem ser de certa forma amenizados dotando-se o sis

tema de elementos para recompilar, se necessário, o programa PLOBAN, em tempo de execução.

A decisão de gerar código Pascal ao invés de, diretamente, produzir-se código executável a partir das consultas LOBAN foi tomada pelos vários motivos apresentados a seguir:

a) tentou-se utilizar peças de software já existentes (ex.: compilador Pascal) para diminuir o esforço de desenvolvimento do pré-processador;

b) buscou-se alcançar uma maior portabilidade do sistema (mais precisamente do pré-processador) já que o programa Pascal gerado pode depois ser executado por qualquer máquina que disponha de compilador desta linguagem e da implementação do método de acesso;

c) já que o programa PLOBAN aceita trechos escritos em Pascal, fatalmente, o compilador Pascal teria que ser invocado de qualquer maneira. Deixando-se para o compilador a tarefa de gerar código executável para todo o programa PLOBAN, evita-se uma duplicação de código e esforço.

7.4.2 Arquitetura do Pré-Processador

A fim de transformar as consultas LOBAN do programa PLOBAN em rotinas Pascal, o pré-processador deve executar uma série de tarefas distintas. Este software foi então projetado como um conjunto de módulos, que interagem entre si, cada um executando uma função específica. A figura 7.3 apresenta, esquematicamente, o pré-processador. Seus módulos são descritos, em mais detalhes, a seguir.

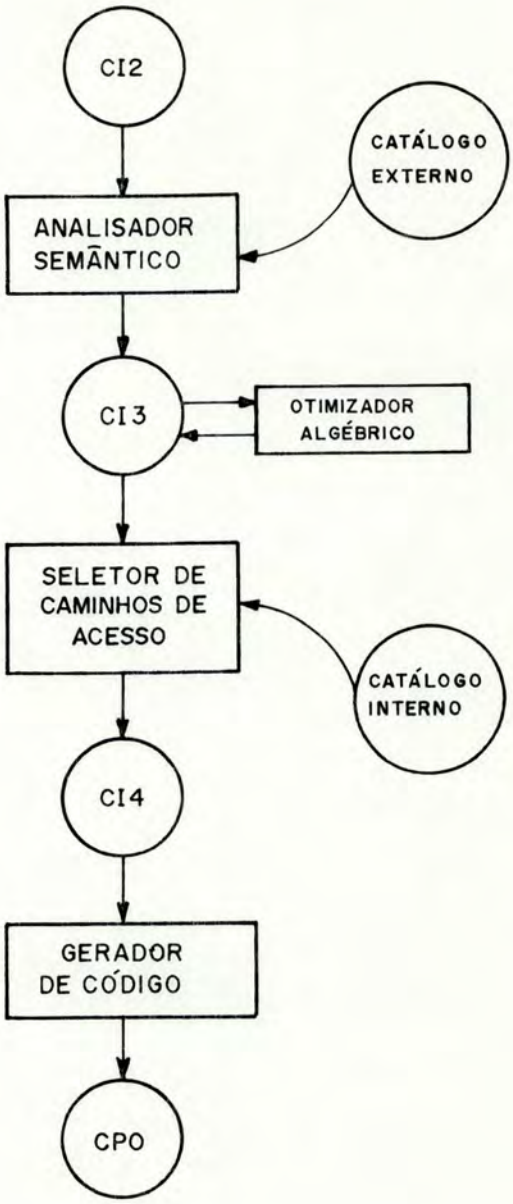

FIGURA 7.3: Arquitetura do Pré-Processador

- *Analisador Semântico*: realiza, principalmente, as funções de resolução de nomes e verificação de compatibilidade entre operandos e operadores [LIM 82a]. Resolução de nomes consiste em traduzir os nomes de construções utilizadas pelo usuário, nas instruções LOBAN, em identificadores da base de dados interna. Para isto, o analisador semântico consulta o Catálogo Externo do sistema que contém a

descrição do banco de dados. A verificação de compatibilidade consiste em assegurar que os tipos de operando estão adequados entre si e, também, em relação ao operador utilizado. Quando tudo está correto, o analisador gera o código intermediário três (CI3). Caso contrário, o processamento do programa PLOBAN é sustado e uma listagem com erros, emitida.

- *Otimizador Algébrico*: este módulo tenta transformar expressões de obtenção de construções com o objetivo de diminuir o volume de dados, que por elas serão manipulados, em tempo de execução [GOL 80]. Um exemplo destas transformações seria a antecipação da execução do operador COLEC (que seleciona tuplas de uma tabela) em uma expressão com outros operadores. Desta forma, os outros operadores, provavelmente, manipularão um conjunto menor de tuplas (somente as que foram selecionadas pelo COLEC). Basicamente, o otimizador executa as operações propostas em [SMI 75]. Este módulo reescreve o CI3 mudando, se possível, a ordem das operações.

- *Seletor de Caminhos de Acesso*: determina o algoritmo mais apropriado para implementar a operação a ser executada. Para esta seleção, são consideradas as estruturas auxiliares existentes (arquivos de inversões) e, no caso de resultados intermediários (dados gerados por um operador e que serão processados por outro dentro da mesma expressão de obtenção de construções), a existência ou não de ordenação [LIM 82b].

A pesquisa de estruturas auxiliares é feita sobre o Catálogo Interno do Sistema que mantém informações sobre as estruturas de dados como estas estão armazenadas em disco.

Outra preocupação deste módulo é em diminuir o número de arquivos intermediários (arquivos que guardam resultados intermediários) procurando realizar o maior núme

ro de operações possíveis sobre cada registro lido (operações embutidas). Assim, o seletor deve decidir quais as operações que produzirão, como resultado, arquivos intermediários (que, possivelmente, serão classificados), e quais "passarão" à próxima operação, um registro de cada vez. Normalmente, o segundo caso é possível quando uma operação aceita, como entrada, registros classificados na mesma ordem que a operação anterior os gerou.

Este módulo gera, como produto de seu processamento, o código intermediário quatro (CI4) que será consumido pelo gerador de código.

- *Gerador de Código*: este módulo gera um programa Pascal a partir do CI4 e dos trechos escritos em Pascal do programa PLOBAN. O gerador de código utiliza-se de uma biblioteca de rotinas Pascal incompletas como base para as rotinas que deve gerar. Estas rotinas são completadas com as informações constantes no CI4. Depois de gerar o programa Pascal completo, este módulo invoca o compilador da linguagem e complementa o código objeto gerado com informações sobre as estruturas de dado manipuladas por este programa e com o CI2. Em tempo de execução, estas informações são consultadas, pelo sistema, para verificar a necessidade ou não de recompilar o programa. O gerador de código é descrito, em maiores detalhes, nas próximas seções.

8. UMA TÉCNICA DE GERAÇÃO DE CÓDIGO PARA LINGUAGENS NÃO-PROCEDURAIS

8.1 Introdução

Nesta seção, é proposta uma nova técnica de geração de código, a partir de linguagens não-procedurais ou híbridas, como é o caso das linguagens de operação de banco de dados, baseadas na abordagem relacional, embutidas em linguagens procedurais de alto nível.

A motivação para o desenvolvimento desta nova técnica surgiu de necessidades específicas do Sistema L para o desenvolvimento de seu módulo gerador de código. Quando da decisão de embutir LOBAN em Pascal, condicionou-se esta tarefa a que o trabalho realizado até o momento não fosse perdido. Além disso, ficou acertado que o código a ser gerado deveria ser o mesmo gerado pelo compilador Pascal da instalação (código P) a fim de ser executado, diretamente, pela máquina P. Observe-se, ainda, que a interface embutida do Sistema L, para a qual deve ser construído o módulo gerador de código, implementa, somente, um subconjunto da linguagem LOBAN, podendo este ser modificado e/ou ampliado conforme se faça necessário.

Considerando-se os fatos e quesitos, descritos acima, projetou-se uma técnica de geração de código que visa atingir, basicamente, quatro objetivos principais:

- manter baixo o custo de desenvolvimento do módulo gerador de código, pela utilização de recursos já existentes na instalação;

- possibilitar a geração de código portável, como é o caso das linguagens de alto nível;

- o módulo gerador de código deve ser capaz de absorver, facilmente, mudanças no conjunto de instruções da

linguagem fonte;

- o código gerado deve ser o mais eficiente possível, levando-se em consideração o compromisso com a portabilidade e com o baixo custo de desenvolvimento do módulo gerador.

Através da técnica criada, traduz-se o programa fonte, em linguagem não-procedural ou híbrida em um programa, escrito em linguagem procedural de alto nível, para a qual exista compilador já implementado. A partir deste código intermediário, o compilador gera código executável.

As seções a seguir descrevem a técnica em maiores detalhes.

8.2 O código de entrada do gerador

A entrada principal do gerador é um código intermediário, de alto nível, na forma de comandos ou grupo de comandos, que manipulam dados ou conjuntos de dados. Este módulo deve, basicamente, traduzir o código de entrada em comandos e/ou grupos de comandos, em linguagem de máquina (real ou virtual) executável.

8.3 Arquitetura do gerador de código

O módulo gerador de código é dividido em três partes, cada uma executando uma função bem específica (figura 8.1).

A PARTE I é responsável pela identificação dos comandos do código intermediário de entrada e por sua posterior tradução para rotinas em linguagem procedural de alto nível, para a qual já se dispõe de compilador.

A fim de executar a tradução dos comandos de entrada, este módulo dispõe de uma biblioteca de rotinas pré-prontas, escritas na linguagem objeto de alto nível. Cada rotina pré-pronta mantém correspondência com um comando de código intermediário. O gerador recebe do seletor de caminhos de acesso a identificação das rotinas a serem preenchidas. As rotinas pré-prontas são assim denominadas, pois devem ser completadas a partir de parâmetros específicos das ocorrências de cada comando do código de entrada do gerador. Estes parâmetros representam as partes variáveis das rotinas; são estruturas de dados e expressões de seleção de estruturas, representando, respectivamente, os operandos e expressões especificadas pelo usuário, nas instruções da linguagem não-procedural.

A PARTE I produz, basicamente, três conjuntos de dados: 1) conjunto das rotinas pré-prontas já preenchidas, que substituem as instruções submetidas pelo usuário ao compilador; 2) conjunto das chamadas a estas rotinas; 3) declarações de estruturas de dados globais às rotinas, que por elas serão manipuladas, em tempo de execução. É através destas estruturas, que as rotinas comunicam-se entre si e com o resto do programa do usuário.

O segundo módulo do gerador de código (PARTE II na figura 8.1) é responsável pela montagem do programa completo, em linguagem procedural de alto nível. Para isto, esta parte intercala as três saídas da PARTE I entre si. Se as instruções sendo compiladas estão embutidas em programa escrito em outra linguagem (linguagem hospedeira), que no caso deve ser a mesma utilizada para escrever as rotinas pré-prontas, a PARTE II deve incluir, também, no processo de intercalação, os trechos do programa do usuário escritos nesta linguagem. A saída da segunda parte é um programa escrito em linguagem procedural de alto nível, para a qual o sistema dispõe de compilador.

A função básica da PARTE III é submeter o progra-

ma gerado na segunda etapa ao compilador da linguagem na qual foi escrito para que este gere, a partir daí, código de máquina executável.

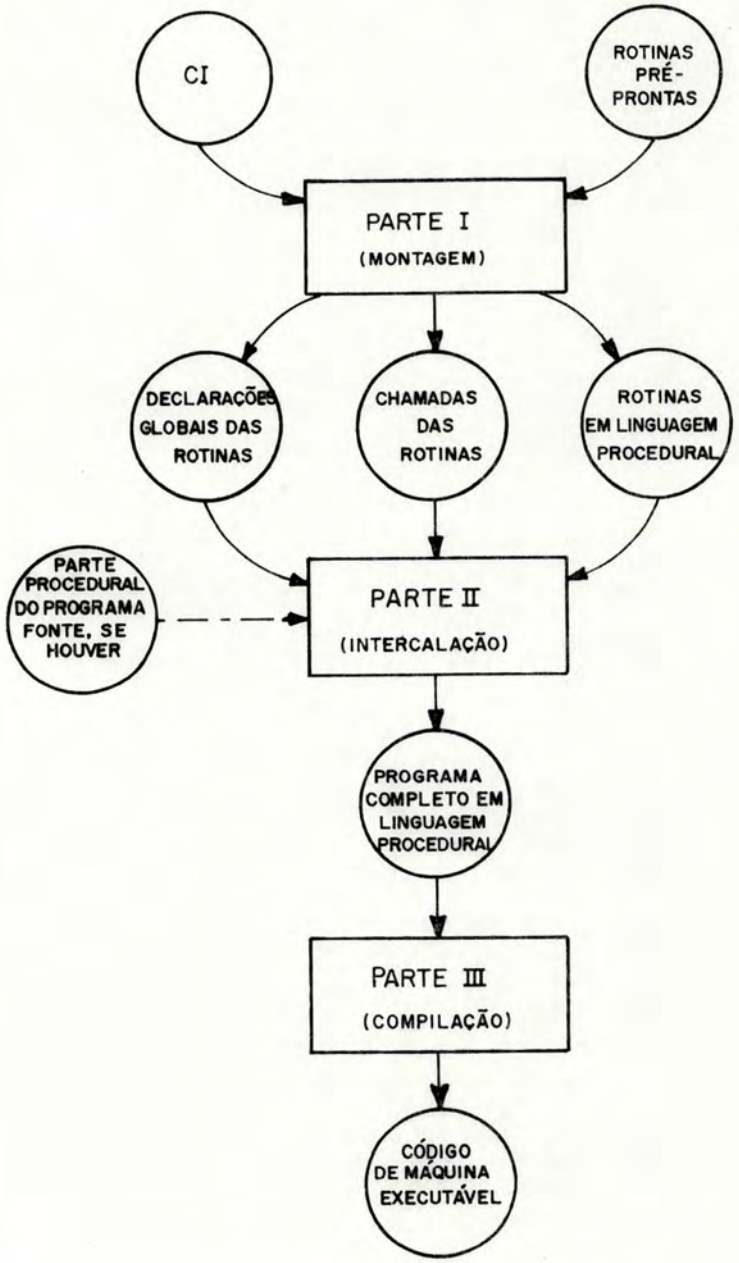

FIGURA 8.1: Arquitetura do Gerador de Código

No caso de linguagens de operação de banco de dados, a PARTE III também pode complementar o arquivo,que contém o código gerado pelo compilador, com informações como data da compilação e as estruturas da base de dados referenciadas pelo programa. Em tempo de execução, estas informações são necessárias para decidir-se se o programa ainda é válido, em relação ao banco de dados, ou deve ser recompilado.

8.4 As rotinas pré-prontas

Na PARTE I do gerador são obtidas as rotinas em linguagem procedural a partir das instruções não-procedurais fornecidas pelo usuário e das rotinas pré-prontas correspondentes.

As rotinas pré-prontas são compostas de trechos, em linguagem procedural e comandos que direcionam as operações para seu preenchimento. Estes comandos apontam, no texto, lugares onde devem ser inseridos parâmetros que estão no código intermediário de entrada do gerador. Além disso, dependendo dos parâmetros, alguns trechos das rotinas poderão ser desprezadas ou, alternativamente, utilizadas. Os comandos aparecem, nas rotinas, sempre delimitados por caracteres especiais reconhecíveis pela PARTE I.

Com os comandos de controle de preenchimento, uma mesma rotina pré-pronta pode gerar rotinas diferentes, dependendo de fatores específicos apresentados por cada ocorrência do comando, ao qual ela está associada, no código intermediário. Um exemplo disso seria a rotina que recupera registros de algum arquivo segundo os valores de um determinado campo. Dependendo do arquivo que deve ser pesquisado, a rotina pode ser preenchida para lê-lo seqüencialmente ou fazer uma pesquisa binária sobre ele ou, ainda, selecionar os registros a partir da pesquisa sobre um arquivo de inver

sões associado a este arquivo de dados. A escolha da estratégia a seguir será feita a partir das informações (parâmetros) que acompanham a ocorrência do comando no código de entrada da PARTE I. Este esquema flexibiliza a tradução dos comandos aumentando, assim, a eficiência do código gerado.

A seguir, é apresentado um exemplo de preenchimento de rotina pré-pronta. Nele aparecem um trecho de código intermediário de entrada do gerador (CI4) e um trecho de rotina pré-pronta (escrita em linguagem procedural, de alto nível, semelhante a Pascal) antes e depois de ser processado pela PARTE I.

O gerador vai transferindo os trechos da rotina pré-pronta, para a área de montagem, até encontrar um caractere especial (que no exemplo é representado pelo caractere '&') que indica o início de um comando a ser executado pelo próprio gerador para definir e montar o próximo trecho da rotina a ser transferido. Estes comandos podem aparecer embutidos em outros.

Exemplo de montagem de rotina pré-pronta (baseado na figura 2.2):

- instrução submetida, pelo usuário, ao compilador (em LOBAN):

  ESTREITAR C EMPREGADOS.TR PARA NOMEEMP,SALARIO

- código intermediário gerado a partir da instrução:

```
ESTREIT (
        ARQENT (NOME=EMPREGADOS.TR; TIPO=RELACIONAL;
                NUMCAMPOS=4; CAMPO1:INTEIRO;
                CAMPO2:CHAR(30); CAMPO3:REAL;
                CAMPO4:INTEIRO)
```

```
        ARQSAI (NOME=RESULTADO; TIPO=SERIAL;  NUMCAMPOS=2;
                CAMPO1:CHAR(30)(ORIGEM=ARQENT.CAMPO2);
                CAMPO2:REAL    (ORIGEM=ARQENT.CAMPO3) )
               );
```

- trecho da rotina pré-pronta, em Pascal, correspondente à instrução ESTREIT do código intermediário:

```
:
& CASO ARQENT.TIPO FOR
     RELACIONAL: "READREL (& INSERE ARQENT.NOME &);";
     SERIAL: "READSER (& INSERE ARQENT.NOME &);";
  END &
WHILE & INSERE ARQENT.NOME &.RESULT <> ENDOFILE
DO  BEGIN
     & MONTA (ARQSAI.TUPLA) &
     WRITE (& INSERE ARQSAI.NOME &);
     END;
:
```

- trecho da rotina pré-pronta depois de processada pela PARTE I do gerador de código:

```
:
READREL (EMPREGADOS.TR);
WHILE EMPREGADOS.TR.RESULT <> ENDOFILE
DO  BEGIN
    WITH  RESULTADO.TUPLA
    DO  BEGIN
        CAMPO1:= EMPREGADOS.TR.CAMPO2;
        CAMPO2:= EMPREGADOS.TR.CAMPO3;
        END;
    WRITE(RESULTADO);
    END;
:
```

9. PROJETO DO MÓDULO GERADOR DE CÓDIGO DO SISTEMA L

9.1 Introdução

O módulo gerador de código do pré-processador da Interface LOBAN Embutida foi projetado com base na técnica de geração apresentada no capítulo 8.

Esta técnica foi adaptada ao ambiente de implementação, o qual apresentava algumas limitações que tiveram de ser levadas em consideração. Por outro lado, observou-se que o próprio ambiente deveria sofrer modificações para que o gerador de código ganhasse em eficiência.

O projeto de implementação do gerador de código estendeu-se por cerca de 12 meses. Durante este tempo sua arquitetura foi modificada 5 vezes (anexo 2). As causas destas modificações foram várias: desconhecimento do funcionamento exato de alguns módulos do pré-processador, ainda não implementados (por exemplo, o analisador semântico e o seletor de caminhos de acesso); modificações na arquitetura do Sistema L e no subconjunto LOBAN a ser, por ele, implementado; limitações do ambiente de implementação (por exemplo, limitações apresentadas pelo SOL e pelo compilador Pascal), e tentativas de evoluir-se para uma arquitetura mais eficiente.

As seções seguintes apresentam, entre outras coisas, as restrições impostas à implementação pelo ambiente,a estrutura do CI4, a arquitetura das partes do gerador para implementação, os comandos de controle das rotinas pré-prontas e as modificações propostas para o ambiente.

9.2 Restrições impostas à implementação

Entre as características apresentadas pelo ambiente de desenvolvimento, três foram as que mais influenciaram o projeto de implementação do gerador de código:

- o Sistema Operacional para LOBAN (SOL) não permite que o tamanho do programa objeto do usuário (que é executado pelo JOBPROCESS) exceda 36 páginas de 512 bytes cada;

- o compilador Pascal da instalação, em sua versão atual, não permite programas fonte com mais de 700 (setecentos) identificadores distintos, entre constantes, tipos de dados, nomes de variáveis e identificadores de rotinas;

- na linguagem Pascal, a declaração das estruturas segue uma ordem rígida. As declarações de constantes, tipos de dados, variáveis e procedimentos (subrotinas) devem aparecer, no programa, nesta ordem. Além disso, as palavras reservadas "CONST", "TYPE" e "VAR" que indicam, respectivamente, o início das declarações de constantes, tipos e variáveis, só podem ocorrer, cada uma, uma única vez no texto fonte.

A partir das duas primeiras restrições, apresentadas acima, decidiu-se que, dependendo do tamanho do programa PLOBAN, o código Pascal a ser gerado seria dividido em um programa seqüencial principal e um ou mais programas,compilados separadamente e chamados, pelo principal, em tempo de execução.

Levando-se em conta que as declarações, em programas Pascal, obedecem a uma ordem rígida e bem definida, optou-se também por utilizar arquivos separados (temporários) para cada tipo de declaração durante a montagem de cada programa seqüencial. Isto facilita o processamento de declara

ções globais das rotinas pré-prontas, que nem sempre ocorrem na ordem esperada pelo compilador Pascal. Estes arquivos temporários são, posteriormente, encadeados na ordem correta de declarações da linguagem.

O tamanho reduzido da memória principal (mais ou menos 13 Kbytes) disponível ao programa seqüencial (do JOBPROCESS) representou, também, uma restrição bastante significativa no projeto do gerador e na arquitetura dos programas a serem gerados.

Outras características do ambiente de implementação, que poderão influenciar, negativamente, na performance do gerador, foram a falta de instruções mais poderosas, em Pascal, para a manipulação de conjuntos de caracteres ("strings") e a baixa performance apresentada pela própria máquina P.

A partir das limitações impostas pelo ambiente de implementação e de algumas decisões, que começaram a ser tomadas para contorná-las, entendeu-se que o projeto deveria refletir um compromisso entre a eficiência do código gerado e a eficiência do próprio módulo gerador. Uma preocupação demasiada com a performance do código a ser gerado poderia levar a uma grande complexidade do gerador, tornando mais difícil seus desenvolvimento e manutenção, além de degradar, muito, sua própria performance. Em contrapartida, o programa gerado deveria apresentar um mínimo de eficiência, em tempo de execução. Em resumo, o usuário deveria receber um código ágil mas não deveria ficar longos minutos, ao lado do computador, esperando que seu programa fosse compilado.

9.3 Estrutura do código intermediário 4 (CI4)

O código intermediário 4 é produzido pelo módulo seletor de caminhos de acesso e contém comandos parametrizados, em linguagem de alto nível. Cada comando do CI4 corresponde a uma das instruções LOBAN do programa PLOBAN. A única instrução LOBAN que dá origem a mais de um comando, neste código, é a instrução de controle de fluxo (FAZER PARA CADA).

Além dos comandos associados às instruções LOBAN, o CI4 contém, no seu final, uma lista gerada pelos analisador semântico e seletor de caminhos de acesso com os nomes externos das estruturas da base de dados interna que serão manipuladas pelo programa PLOBAN.

Também aparecem, no código intermediário 4, a identificação do arquivo em disco que contém o programa PLOBAN e marcas indicativas associadas ao início de trechos, escritos em Pascal, do mesmo programa. Estes indicadores são denominados "marcas Pascal". Estas informações são importantes durante a fase de intercalação do gerador.

Os comandos e "marcas Pascal" do CI4 estão dispostos na mesma ordem relativa em que as suas instruções associadas e trechos escritos em Pascal aparecem no programa PLOBAN.

Os comandos do CI4 apresentam a seguinte forma geral:

```
COMANDO(ENTRADA1, ENTRADA2, SAÍDA, ESTRUTURAS AUXILIARES,EX
        PRESSÕES BOOLEANAS)
```

Os comandos representam nodos em árvores de operação. Suas entradas representam seus nodos filhos na árvore e podem ser tanto arquivos como outros comandos(denominados comandos embutidos). Todos os comandos têm, pelo menos, uma entrada.

A especificação de saída fornece informações sobre o tipo e o formato do resultado a ser produzido pelo comando.

As áreas auxiliares são utilizadas para armazenar dados temporários e as expressões booleanas podem representar expressões de seleção (ex.: comando COLEC).

A seguir, é apresentada a forma geral da sintaxe do CI4, utilizando-se para isto a BNF estendida da tabela 2.1. O anexo 3 apresenta a sintaxe completa deste código.

```
<CI4> ::= <id.arq.PLOBAN>(<comando>|<marca Pascal>...)<lista
                                                de estruturas>
   <id.arq.PLOBAN>::=<área do disco><arquivo da área>
   <comando>::= <id.comando>(<parâmetro>...)
         <id.comando>::= <classe do comando><tamanho código>
                         <id.arq.rotina pré-pronta>
              <classe do comando>::= INÍCIO FAZER PARA CADA(Ø) |
                                     FIM FAZER PARA CADA(1) |
                                     OPERATIVO(2)
              <tamanho código>::=<valor inteiro>
              <id.arq.rotina pré-pronta>::=<área do disco>
                                           <arquivo da área>
         <parâmetro>::= <comando>|<arquivo>|
                        <expressão de seleção>|
                        <estrutura auxiliar>
   <lista de estruturas>::= (<nome externo de arquivo>,,,)
```

O atributo de comando <tamanho código> é um número que representa o tamanho em caracteres do código gerado pela compilação da rotina pré-pronta associada ao comando mais os códigos gerados a partir das rotinas pré-prontas associadas a seus comandos embutidos como parâmetros. É importante observar que o tamanho do código gerado a partir da maior rotina pré-pronta não excede o tamanho máximo de um

programa seqüencial.

Os dois primeiros parâmetros de um comando (em ordem de gravação) são os únicos que podem ser do tipo comando. Isto se deve ao fato de a árvore de operações ser sempre binária. Cada operador (comando) tem, no máximo,dois arquivos de entrada (ex.: operador JUNTAR tem duas entradas; operador ESTREITAR tem uma entrada). Os arquivos de entrada podem ser reais (da base de dados ou canal auxiliar) ou virtuais (dados resultantes de operações embutidas ao comando).

Os comandos do código intermediário 4 estão em notação pré-fixada, em relação a seus parâmetros. Contudo, a geração de código é feita primeiro para os operandos e, após, para os operadores. Esta estratégia visa alcançar três objetivos:

- a segmentação de comandos cujo tamanho do código total a ser gerado excede o tamanho máximo de um programa seqüencial é feita, analisando-se a árvore de operações (formada por eles e seus comandos embutidos) de cima para baixo. Com isto, normalmente, a divisão da árvore de operações com a conseqüente criação de arquivos intermediários (criados pelas subárvores de baixo e consumidos pela de cima) acontecerá em pontos onde o número de tuplas a processar já foi reduzido pelas operações abaixo, resultando em arquivos intermediários menores;

- a decisão de segmentar um comando e sua posterior segmentação acontecem, sempre, antes que qualquer processamento sobre ele tenha sido iniciado;

- o processamento, de baixo para cima, na árvore de operações garante que o preenchimento da rotina pré-pronta, associada ao comando da raiz, só é iniciado quando as rotinas e variáveis globais, que serão por ele utilizadas, já tiverem sido declaradas.

A figura 9.1 ilustra um exemplo de árvore de operação que deve ser segmentada. Através deste exemplo, pode-se perceber a vantagem de um algoritmo de segmentação que analisa a árvore de forma pré-fixada.

O tamanho total do código a ser gerado, a partir dos comandos da árvore, é 40 (10 do JUNTAR1 mais 30 de seus comandos embutidos). Como o tamanho máximo de programas seqüenciais é 30, deve-se dividir a árvore de operações em subárvores que geram código de tamanho igual a este valor, ou menor. Se a árvore fosse pesquisada de forma pós-fixada ou infixada (visitando primeiro o filho à esquerda), esta seria segmentada no ponto 1 da figura. No caso de uma pesquisa de forma pré-fixada porém, a árvore seria dividida no ponto 2. O número de tuplas geradas pela operação JUNTAR2 será sempre igual ou maior do que o total de tuplas selecionadas pela operação COLEC. Sendo assim o ponto mais adequado para a inserção de um arquivo intermediário é o 2.

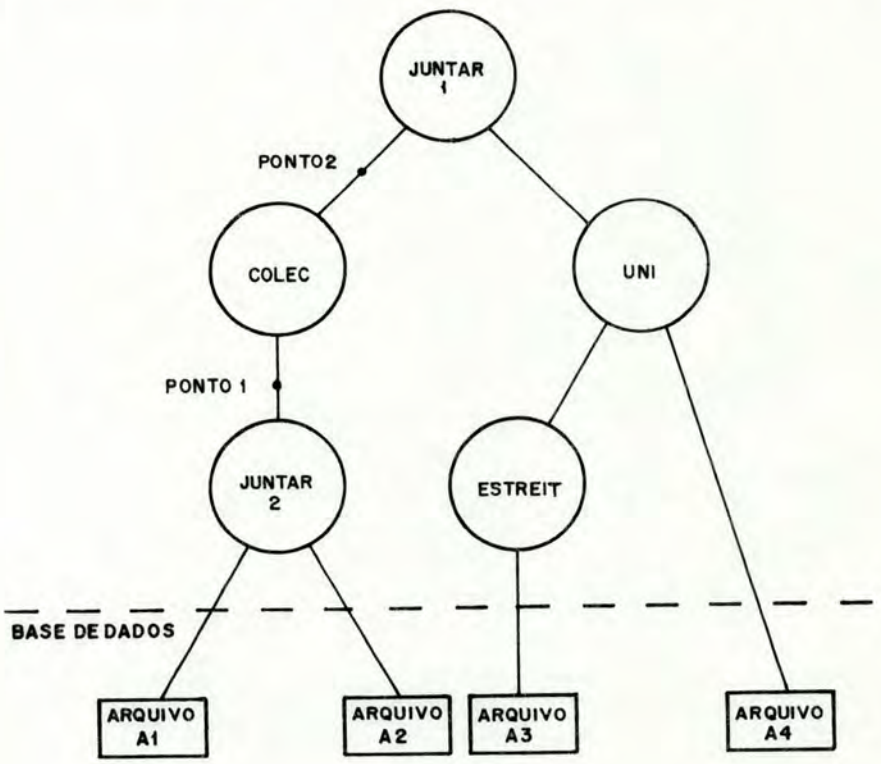

FIGURA 9.1 : Exemplo de Segmentação em Árvore de Operação

9.4 Arquitetura do módulo gerador de código para implementação

9.4.1 Introdução

Com base nas decisões tomadas, a partir da análise das características do ambiente de implementação, chegou-se a, pelo menos, duas alternativas para o projeto da arquitetura do gerador a ser implementado:

- implementação síncrona: o gerador executaria suas três fases seqüencialmente. Em um primeiro momento, a PARTE I varreria todo o CI4 montando todos os programas seqüenciais com todos os seus arquivos temporários. Logo após, a PARTE II geraria um a um os códigos Pascal dos programas a serem compilados, a partir de seus respectivos arquivos temporários. Por fim, a PARTE III invocaria o compilador Pascal para cada um dos programas fonte gerados pela PARTE II;

- implementação assíncrona: a PARTE I executa concorrentemente com o complexo formado pelas partes II e III. Cada vez que a PARTE I termina a montagem dos arquivos temporários de um programa seqüencial, estes são intercalados pela PARTE II e o programa Pascal gerado é, imediatamente, compilado.

A principal vantagem da abordagem síncrona é que sua implementação não é muito complexa pois a comunicação entre os diversos módulos do gerador se dá, unicamente,através dos arquivos temporários existentes, a cada momento, em disco. Por outro lado, a compilação dos programas PLOBAN pode ficar muito demorada e o número de arquivos temporários, criados pela PARTE I, pode tornar-se proibitivo.

Na abordagem assíncrona, o tempo de processamento é menor, assim como o número de arquivos temporários existentes a cada momento no disco (os arquivos podem ser reu-

sados pela PARTE I depois de submetidos às partes II e III). Esta opção representa, porém, um aumento na complexidade da implementação. Deve-se criar mecanismos para a sincronização dos processos.

Depois de um estudo de viabilidade, em relação ao sistema SOL, decidiu-se adotar o esquema de execução assíncrona para diminuir o tempo de processamento total, aproveitando-se, para isto, o ambiente de multiprogramação oferecido pelo sistema operacional, com algumas modificações. A decisão de executar as partes II e III sincronamente e, portanto no mesmo processo, foi tomada pois a troca de processos, feita pelo SOL também é relativamente lenta. Além disso, o código Pascal destas partes é grande e a maioria dos processos do SOL só podem executar programas seqüenciais pequenos. Atualmente, só o JOBPROCESS poderia manter estes programas.

No decorrer do projeto verificou-se que o usuário PLOBAN poderia ter problemas em entender seu programa após a fase de pré-processamento. O fato de o sistema definir novas variáveis e rotinas poderia, facilmente, confundir o usuário se, por exemplo,ocorresse algum problema de lógica em seu programa. O número original de cada linha fonte do programa PLOBAN seria alterado na geração do programa Pascal, além de aparecerem uma série de novas declarações de rotinas e variáveis. As instruções LOBAN escritas pelo usuário dariam lugar, no texto fonte, a chamadas a rotinas escritas em Pascal.

A solução que os projetistas do Sistema R[CHA 81] encontraram, para o problema acima, foi a substituição das instruções em SQL por chamadas a módulos compilados como rotinas externas ao programa. Com isto as declarações, feitas pelo sistema, no programa do usuário, diminuiriam significativamente além de o número de seqüência de cada linha fonte permanecer, praticamente, o mesmo.

No caso do Sistema L esta solução também é viável. O código a partir de cada instrução LOBAN poderia ser gerado como um programa seqüencial, em separado, que posteriormente seria invocado pelo programa principal criado a partir dos trechos, em Pascal, do programa PLOBAN do usuário.

O único caso ao qual esta técnica não pode ser aplicada é o da instrução de controle de fluxo (FAZER PARA CADA) que pode conter, no seu bloco de comandos, instruções Pascal. A execução de instruções Pascal fora do programa principal exigiria um mecanismo complexo e, certamente, lento para a identificação e passagem de parâmetros. Também não seria razoável, em termos de performance, dividir a instrução iterativa, causando-se assim a troca de programas a cada iteração.

Optou-se por uma solução intermediária. Os códigos de todas as instruções LOBAN, que não forem de controle de fluxo, serão gerados como programas seqüenciais em separado. As instruções "FAZER PARA CADA" serão expandidas no próprio programa principal assim como as instruções LOBAN que serão executadas dentro de seus blocos de comandos.

9.4.2 Apresentação da arquitetura do gerador

A figura 9.2 apresenta, na forma de uma rede de estações e canais, a arquitetura do módulo gerador de código do Sistema L, a ser implementado. Ela reflete, basicamente, a arquitetura apresentada na seção anterior com as adaptações que se fizeram necessárias e que foram comentadas acima.

O módulo montador (PARTE I) executa concorrentemente com os módulos intercalador (PARTE II) e compilador (PARTE III). Estas duas últimas partes do gerador funcionam sincronizadas entre si (em um mesmo processo do SOL).

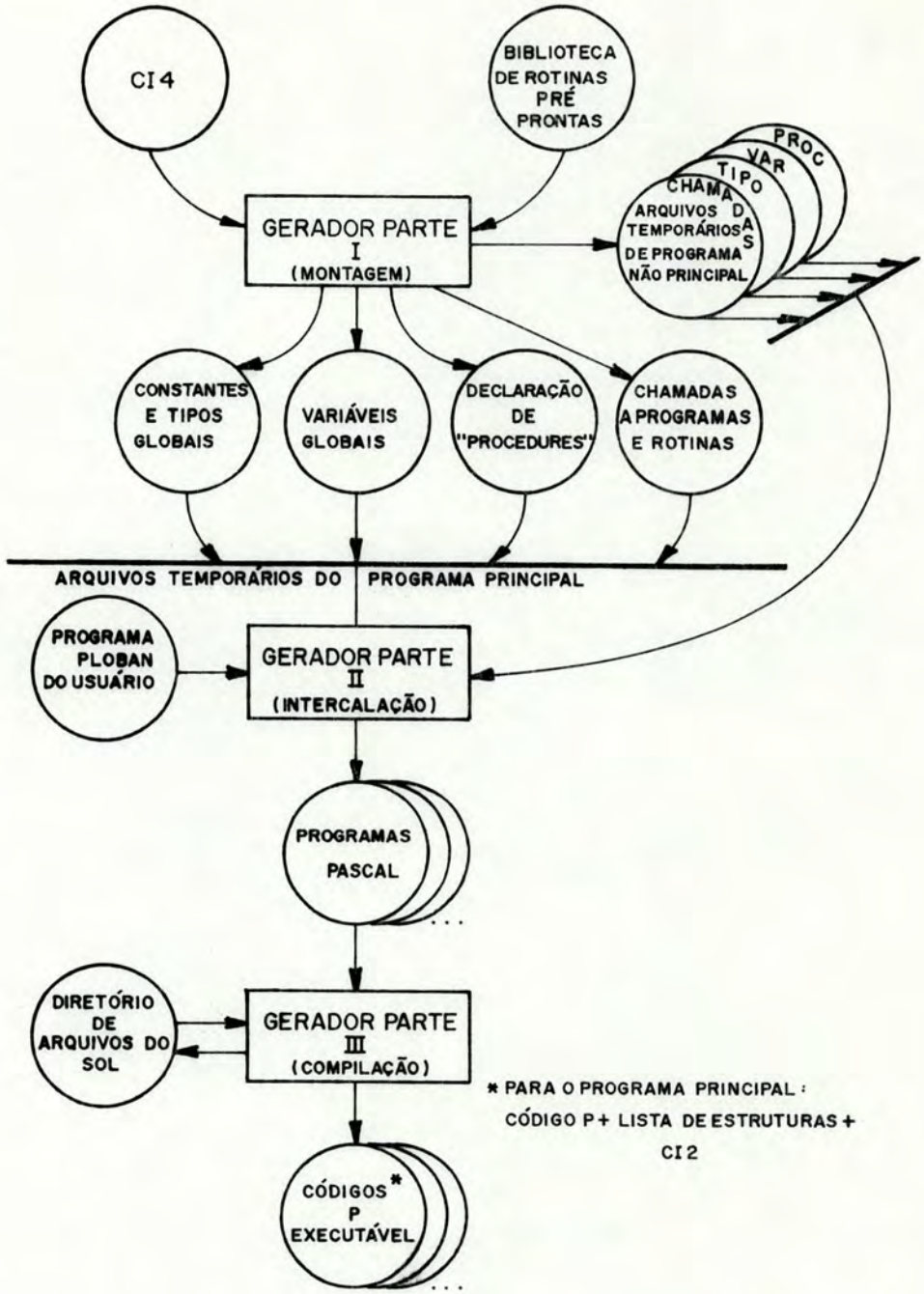

FIGURA 9.2: Arquitetura do Gerador de Código para Implementação

Cada programa seqüencial a ser gerado deve dispor de um conjunto de quatro arquivos temporários. O primeiro que armazenará as declarações de constantes e tipos; o segundo abrigará as variáveis globais do programa; no terceiro deverão ser gravadas as declarações de procedimentos

Pascal obtidos através do preenchimento das rotinas pré-prontas; o último arquivo temporário, de cada grupo, conterá as chamadas aos procedimentos (e a outros programas seqüenciais, assim como as marcas Pascal encontradas no CI4, quando se tratar do programa principal).

Deverá existir um grupo de arquivos temporários alocado, durante todo o tempo de execução da PARTE I, para o programa principal a ser gerado. O número ótimo de grupos de arquivos temporários para a geração dos outros programas seqüenciais só poderá ser determinado após a implementação do gerador, quando se puder verificar a relação entre os tempos de execução da PARTE I e das partes II e III.

No início do processamento de cada comando do CI4 (que não seja de controle de fluxo) o módulo montador requisita, através de um monitor (para sua comunicação com a PARTE II), um conjunto de arquivos para gerar um programa seqüencial.

Após o preenchimento dos arquivos, a PARTE I torna-os disponíveis ao módulo intercalador, por meio do mesmo monitor. Este módulo, quando está ocioso, requisita, através deste monitor, um grupo de arquivos prontos para a intercalação. Após gerar o programa Pascal a partir dos arquivos temporários, a PARTE II libera-os através do mesmo monitor e invoca a execução da PARTE III que compilará este programa e concatenará o código gerado à lista de estruturas e ao CI2 formando, assim, o canal programa objeto.

No final do processamento do CI4, a PARTE I libera o grupo de arquivos temporários do programa principal para que estes sejam intercalados, gerando um programa Pascal que é, logo após, compilado. Esta atividade marca o fim da execução do módulo gerador de código.

9.4.3 Estrutura do módulo montador (PARTE I)

9.4.3.1 Introdução

Nesta seção, o módulo montador do gerador de código é apresentado em maiores detalhes. Apresenta-se os seus algoritmos de funcionamento e as estruturas de dados auxiliares que utiliza.

A descrição do monitor, que este módulo utiliza para comunicar-se com a PARTE II, será apresentada em seção posterior.

9.4.3.2 Algoritmos de funcionamento da PARTE I

- *Varredura dos comandos do CI4 (laço externo):*

```
1 - TEM-LUGAR-NO-PROG-PRINCIPAL:= VERDADEIRO;
2 - CONTADOR-FAZER:= 0;
3 - PARTEI.ESTADO:= NORMAL;
4 - PARA CADA REGISTRO DO CI4 (até o início da lista de es-
                               truturas)
    FAZER
        4.1 - CASO O TIPO DO REGISTRO
              FOR
                 4.1.1 - MARCA-PASCAL:
                         - GRAVA A INFORMAÇÃO NO ARQUIVO DE
                           CHAMADAS DO PROGRAMA PRINCIPAL;
                 4.1.2 - COMANDO:
                         - EXECUTA A ROTINA "TRATA-COMANDO-
                           CI4";
    FIM-FAZER;
```

- *Trata Comando CI4:*

```
1 - CASO A CLASSE DO COMANDO
    FOR:
        1.1 - INÍCIO-FAZER-PARA-CADA:
              - CONTADOR-FAZER:= CONTADOR-FAZER + 1;
              - PARTEI.ESTADO:= FAZER-PARA-CADA;
              - EXECUTA A ROTINA "TRATA-INÍCIO-FAZER-PARA-
                CADA";

        1.2 - FIM-FAZER-PARA-CADA:
              - EXECUTA A ROTINA "TRATA-FIM-FAZER-PARA-CADA";
              - SE(CONTADOR-FAZER:= CONTADOR-FAZER - 1 = 0)
                ENTÃO PARTEI.ESTADO:= NORMAL;

        1.3 - COMANDO-OPERATIVO:
              - SE PARTEI.ESTADO = FAZER-PARA-CADA
                ENTÃO  EXECUTA A ROTINA "TENTA-CRIAR-ROTINA-
                                        NO-PROG-PRINCIPAL"
                SENÃO EXECUTA A ROTINA "GERA-PROGRAMA-SEQÜENCIAL";
```

- *Trata Início Fazer Para Cada:*

```
1 - MONTA NO ARQUIVO DE CHAMADAS DO PROGRAMA PRINCIPAL    O
    INÍCIO DO LAÇO DO COMANDO;
2 - ATUALIZA TABELA DE PONTOS CORRENTES(TABPC);
```

- *Trata Fim Fazer Para Cada:*

```
1 - MONTA NO ARQUIVO DE CHAMADAS DO PROGRAMA PRINCIPAL    O
    FIM DO LAÇO DO COMANDO;
2 - DELETA A ÚLTIMA ENTRADA NA TABELA DE PONTOS   CORRENTES
    (TABPC);
```

- *Tenta Criar Rotina no Prog. Principal:*

```
1 - SE TEM-LUGAR-NO-PROG-PRINCIPAL
    ENTÃO FAÇA:
          1.1 - SE TAMANHO-ATUAL + TAMANHO-CÓDIGO (COMANDO)
                                   >
                              TAMANHO-MÁXIMO
                ENTÃO EXECUTA A ROTINA "GERA-PROGRAMA-SEQUENCIAL"

          1.2 - SENÃO FAÇA:
                - TAMANHO-ATUAL:=
                   TAMANHO-ATUAL + TAMANHO-CÓDIGO(COMANDO);
                - SE(TAMANHO-MÁXIMO) - (TAMANHO-ATUAL)
                                   <
                              TAMANHO-MÍNIMO
                  ENTÃO TEM-LUGAR-NO-PROG-PRINCIPAL := FALSO;
                - EXECUTA A ROTINA "PREENCHE-ROTINA-PRÉ-PRONTA"
                  USANDO O GRUPO DE ARQUIVOS DO PROGRAMA
                  PRINCIPAL;
                FIM-FAÇA;
          FIM-FAÇA

2 - SENÃO EXECUTA A ROTINA "GERA-PROGRAMA-SEQUENCIAL";
```

- *Gera Programa Seqüencial:*

1 - SE TAMANHO-CÓDIGO(COMANDO) > TAMANHO-MÁXIMO
ENTÃO EXECUTA A ROTINA "SEGMENTA-COMANDO"

2 - SENÃO FAÇA:
- REQUISITA GRUPO DE ARQUIVOS TEMPORÁRIOS ATRAVÉS DO MONITOR;
- EXECUTA A ROTINA "PREENCHE-ROTINA-PRÉ-PRONTA" USANDO O GRUPO DE ARQUIVOS TEMPORÁRIOS;
- GRAVA UMA CHAMADA AO PROGRAMA NO ARQUIVO DE CHAMADAS DO PROGRAMA PRINCIPAL;
- LIBERA O CONJUNTO DE ARQUIVOS TEMPORÁRIOS PARA A PARTE II ATRAVÉS DO MONITOR;

FIM-FAÇA;

- *Preenche Rotina Pré-pronta:*

1 - EXECUTA ROTINA "DECLARAÇÕES-GLOBAIS";
2 - ABRE ARQUIVO DA ROTINA PRÉ-PRONTA INDICADO NO COMANDO;
3 - PARA CADA CARACTERE LIDO DO ARQUIVO
FAÇA:
3.1 - SE É SÍMBOLO DE INÍCIO DE FUNÇÃO
ENTÃO EXECUTA ROTINA "PROCESSA-FUNÇÃO"
3.2 - SENÃO TRANSFERE CARACTER PARA O ARQUIVO TEMPORÁRIO DE DECLARAÇÕES DE ROTINAS;
FIM-FAÇA;
4 - GRAVA UMA CHAMADA DA ROTINA NO ARQUIVO DE CHAMADAS DO GRUPO DE ARQUIVOS TEMPORÁRIOS SENDO UTILIZADO;

- *Declarações Globais:*

```
1 - VARIÁVEL-BOOLEANA:= VERDADEIRO;

2 - ENQUANTO HOUVER PARÂMETROS DO COMANDO NO CI4 E
              VARIÁVEL-BOOLEANA
    FAÇA:
         2.1 - RECUPERE O PARÂMETRO;
         2.2 - CASO O TIPO DO PARÂMETRO
               FOR:
                    2.2.1 - COMANDO:
                            - EXECUTA A ROTINA
                              "PREENCHE-ROTINA-PRÉ-PRONTA"
                              USANDO O GRUPO DE ARQUIVOS
                              ATUAL;
                    2.2.2 - ARQUIVO:ESTRUTURA-AUXILIAR:
                            - DECLARA NORMALMENTE;
                    2.2.3 - EXPRESSÃO-DE-SELEÇÃO:
                            - VARIÁVEL-BOOLEANA:= FALSO;
    FIM-FAÇA;
```

- *Segmenta Comando:*

1 - SE(O PRIMEIRO PARÂMETRO DO COMANDO É TAMBÉM UM COMANDO) E
(TAMANHO-CÓDIGO(COMANDO) - TAMANHO-CÓDIGO(PRIMEIRO PARÂMETRO)
<= TAMANHO-MÁXIMO)

ENTÃO FAÇA:
- EXECUTA A ROTINA "SUBSTITUI-COMANDO-EMBUTIDO"
USANDO O PRIMEIRO PARÂMETRO;
- EXECUTA A ROTINA "GERA-PROGRAMA-SEQÜENCIAL"
USANDO O COMANDO;

FIM-FAÇA

2 - SENÃO SE(O SEGUNDO PARÂMETRO DO COMANDO É TAMBÉM UM COMANDO) E
(TAMANHO-CÓDIGO(COMANDO) - TAMANHO-CÓDIGO(SEGUNDO
PARÂMETRO)
<= TAMANHO-MÁXIMO)

ENTÃO FAÇA:
- EXECUTA A ROTINA "SUBSTITUI-COMANDO-EMBUTIDO"
USANDO O SEGUNDO PARÂMETRO;
- EXECUTA A ROTINA "GERA-PROGRAMA-SEQÜENCIAL"
USANDO O COMANDO;

FIM FAÇA

SENÃO FAÇA:
- EXECUTA A ROTINA "SUBSTITUI-COMANDO-EMBUTIDO"
USANDO O PRIMEIRO PARÂMETRO;
- EXECUTA A ROTINA "SUBSTITUI-COMANDO-EMBUTIDO"
USANDO O SEGUNDO PARÂMETRO;
- EXECUTA A ROTINA "GERA-PROGRAMA-SEQÜENCIAL"
USANDO O COMANDO;

FIM-FAÇA;

- *Substitui Comando Embutido:*

1 - MONTA O PARÂMETRO COMO UM COMANDO À PARTE QUE PRODUZ UM ARQUIVO INTERMEDIÁRIO;

2 - EXECUTA A ROTINA "GERA-PROGRAMA-SEQÜENCIAL" USANDO O COMANDO RECÉM CRIADO;

3 - SUBSTITUI O PARÂMETRO DO COMANDO DO CI4 PELO IDENTIFICADOR DO ARQUIVO INTERMEDIÁRIO A SER GERADO;

O algoritmo da rotina "PROCESSA-FUNÇÃO" não será apresentado aqui. Basicamente, ele identifica que função deve ser processada e desvia o controle para uma rotina que a processa.

Existe uma série de funções que podem ser invocadas a partir de rotinas pré-prontas. Elas serão apresentadas em seção a parte.

Muitas funções buscam informações em tabelas mantidas pelo módulo montador (PARTE I). Estas tabelas são descritas a seguir.

9.4.3.3 Estruturas auxiliares da PARTE I:

O módulo montador do gerador de código dispõe de 4 tabelas onde mantém a informação necessária ao preenchimento das rotinas pré-prontas:

- tabela de informação de registros e filemaps (TIR);
- tabela de associação de arquivos e listas de atributos (TACLA);
- tabela de controle de pontos correntes (TABPC);
- tabela de Arquivos Intermediários (TAI).

A seguir, cada uma das tabelas acima é descrita em termos de suas estrutura e funcionalidade.

9.4.3.3.1 Tabela de Informação de Registros e Filemaps (TIR)

Esta tabela armazena os atributos dos parâmetros de um comando do CI4. Suas entradas são preenchidas durante a execução da rotina "Declarações Globais". A cada execução desta rotina, uma ocorrência da TIR é criada no HEAP do programa [CAR 82].

As entradas da TIR são preenchidas, em ordem crescente, conforme os parâmetros do comando vão sendo lidos do CI4. As funções das rotinas pré-prontas referenciam os parâmetros, do comando sendo processado, pelos números das suas entradas nesta tabela. Estes números refletem a ordem relativa dos parâmetros no CI4.

A figura 9.3 apresenta uma visão esquemática da TIR. A estrurura de suas entradas é apresentada como uma declaração de tipo em Pascal.

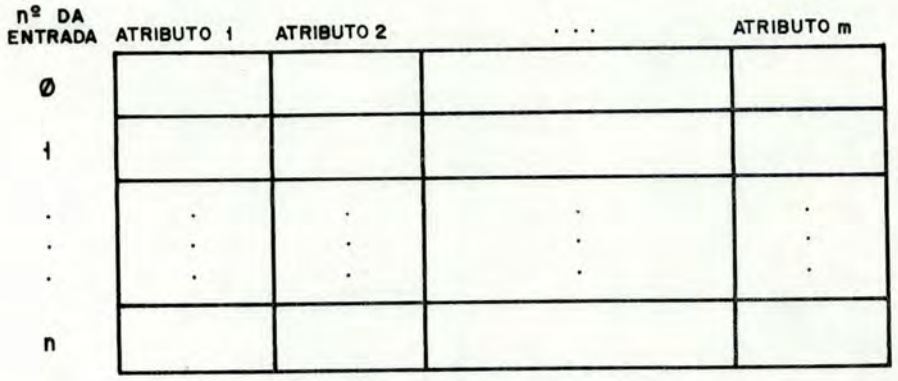

FIGURA 9.3: Visão Esquemática da TIR

```
TYPE ENTRADA-NA-TIR = RECORD
                        NUMERO-CAMPOS-REG,
                        SUFIXO-ID-REG,
                        TIPO-PRIMEIRO-CAMPO:INTEGER;
                        CASE TIPO-REG:INTEGER OF
                           TUPLA,LIGAÇÃO:
                                (TIPO-ARQUIVO,
                                 AREANO,FILENO,
                                 NÚMERO-CAMPOS-LIG: INTEGER);
                           LISTA-DETALHADA-DE-ATRIBUTOS
                                (TIPO-LDA,ÍNDICE-TACLA:INTEGER)
                      END;
```

Quando a rotina "Declarações Globais" identifica um parâmetro do tipo arquivo e este é um arquivo da base de dados, canal auxiliar ou intermediário, ela declara uma área de memória para abrigar suas tuplas ou ligações e uma área de memória para armazenar os dados de seu descritor também chamado "FILEMAP"[CAR 82]. Os identificadores destas áreas têm, como sufixos, números de três algarismos,obtidos de um contador global que é incrementado após cada declaração destes pares de áreas. Como o prefixo de identificadores de registros é diferente daquele usado para identificar filemaps (MBR e MBF respectivamente), não existe o risco de identificar-se duas áreas com um mesmo identificador.

O sufixo dos identificadores é guardado na TIR (atributo SUFIXO-ID-REG) e é usado para substituir as referências às entradas na TIR por seus identificadores completos, durante o preenchimento das rotinas pré-prontas.

O atributo TIPO-PRIMEIRO-CAMPO é utilizado por funções de inicialização de registros.

No caso de o registro ser de ligações o campo NUMERO-CAMPOS-REG contém o número de campos dos registros ligantes e o campo NUMERO-CAMPOS-LIG, o número de campos

dos registros ligados.

O atributo TIPO-REG indica se a entrada em questão representa um arquivo relacional, uma tabela ligacional ou, ainda, uma lista detalhada de atributos.

As listas detalhadas de atributos (LDAs) são estruturas auxiliares que ocorrem como parâmetros de comandos no CI4. Sua estrutura é semelhante a de um registro, porém mantém também informações sobre a origem dos valores de cada um de seus campos. As LDAs podem ser utilizadas de duas maneiras: representando a chave primária ou de comparação dos registros de um arquivo, ou indicando a composição de uma tupla ou ligação de saída do comando do qual é parâmetro.

Quando a rotina "Declarações Globais identifica um parâmetro do tipo LDA, ela declara uma área de memória que conterá seus campos, em tempo de execução, e aloca uma entrada na TACLA onde armazena os dados sobre a origem dos mesmos.

O campo TIPO-LDA indica se a estrutura representa uma tupla ou uma ligação. O ÍNDICE-TACLA indica qual a entrada, daquela tabela, que contém informações sobre a origem dos campos da LDA.

9.4.3.3.2 Tabela de Associação de Arquivos e Listas de Atributos (TACLA)

A TACLA evoluiu de uma tabela para uma estrutura de dados mais complexa. Por motivos históricos, decidiu-se manter sua denominação original.

Esta estrutura armazena as informações de origem dos campos das listas detalhadas de atributos. Cada entrada da TACLA contém os dados de uma LDA.

Uma ocorrência da TACLA é criada no Heap, a cada execução da rotina "Declarações Globais". No final de cada execução da rotina "Preenche Rotina Pré-pronta", a PARTE I deleta as ocorrências mais recentes das TIR e TACLA, liberando suas áreas no Heap do programa para futura reutilização.

Como cada LDA pode ter um número qualquer de campos, a TACLA foi projetada como um arranjo de ponteiros. Cada ponteiro endereça uma estrutura encadeada, criada dinamicamente, com informações sobre os campos de uma LDA. O primeiro nodo de cada estrutura contém o número de campos da LDA, se esta representar uma tupla, ou o número de campos da tupla de ligantes que segue, caso ela represente uma ligação. No caso da LDA representar uma tupla, os próximos nodos da estrutura encadeada mantêm, cada um, a informação de origem de um de seus campos. Se for uma ligação, após o último nodo com informações sobre a tupla de ligantes, segue uma nova estrutura com o número de campos e as informações da tupla de ligados.

Para facilitar o acesso direto às informações sobre as tuplas de ligados das LDAs, a TACLA dispõe de outro arranjo de ponteiros que endereçam, cada um, o nodo com o número de campos da tupla de ligados de cada LDA. Caso a LDA não represente uma ligação, o ponteiro do segundo arranjo a ela associado, terá valor "NIL", significando que não foi inicializado.

A figura 9.4 apresenta uma visão esquemática de ocorrência da TACLA.

Os nodos da estrutura encadeada, que mantêm informações da origem dos campos das LDAs, têm a seguinte estrutura:

```
TYPE ORIGEM-CAMPO-LDA = RECORD
                          ÍNDICE-DA-ORIGEM-NA-TIR,
                          NÚMERO-DO-CAMPO-NA-ORIGEM,
                          ORDENAÇÃO-DO-CAMPO : INTEGER
                        END
```

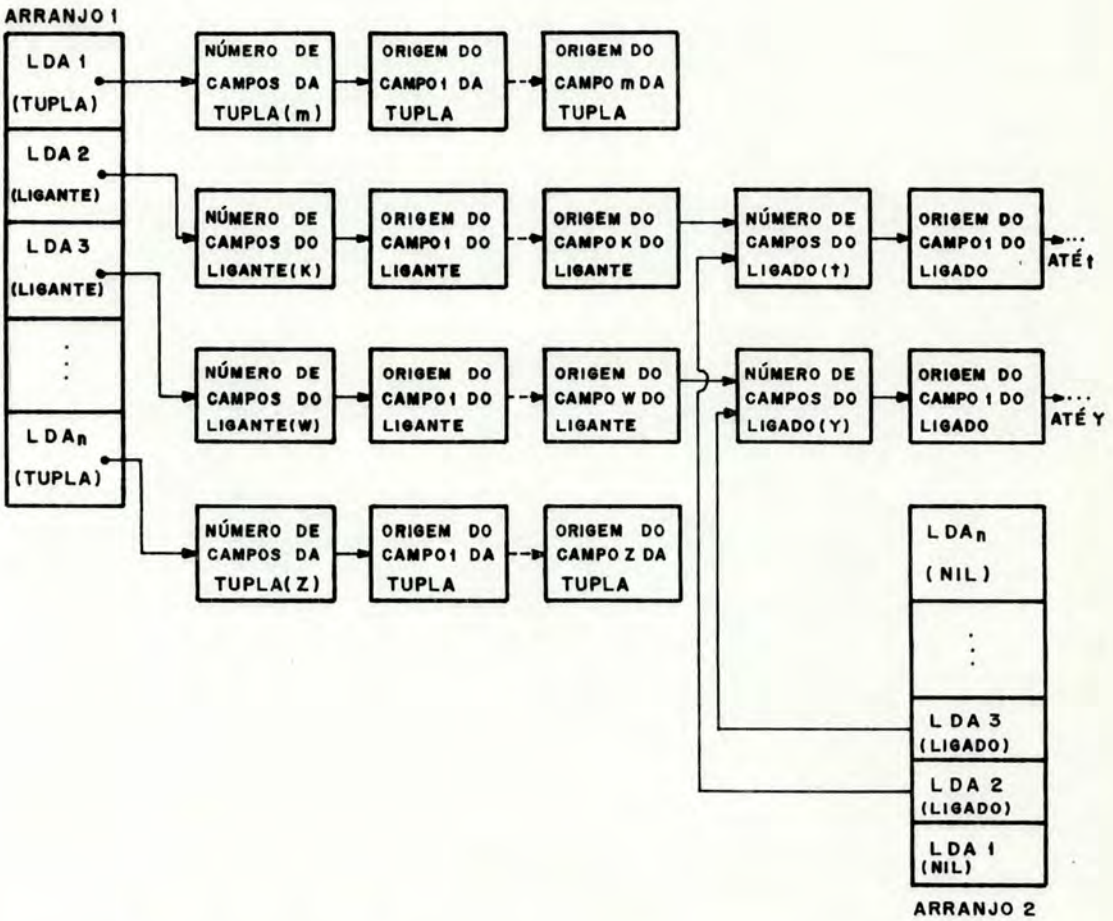

FIGURA 9.4: Visão Esquemática da Tacla

A informação ÍNDICE-ORIGEM-NA-TIR informa o arquivo de origem do campo da LDA. O NÚMERO-DO-CAMPO-NA-ORIGEM indica qual o campo do registro deste arquivo que está associado ao campo da LDA em questão. A ORDENAÇÃO-DO-CAMPO é um dado importante utilizado pelas funções que implementam trechos, em Pascal, de comparação entre campos, durante o

processamento de rotinas pré-prontas.

9.4.3.3.3 Tabela de Pontos Correntes (TABPC)

A instrução LOBAN de controle de fluxo (FAZER PARA CADA) executa um bloco de instruções LOBAN e Pascal para cada construção endereçada da base de dados|canal auxiliar ou do resultado da execução de uma expressão de obtenção de construções. O conjunto destas construções é chamado Conjunto Controlador.

A construção obtida no início de cada iteração do comando FAZER PARA CADA é denominada Ponto Corrente e é representada, em LOBAN, pelo identificador "PC". As instruções LOBAN do bloco iterativo podem fazer referência ao PC (ex.: INCLUIR PC EM EMPREGADOS.TR).

Como pode haver aninhamento de comandos de controle de fluxo (comandos embutidos) as instruções dos blocos iterativos podem referenciar mais de um PC. Sendo assim a linguagem permite associar, aos comandos FAZER PARA CADA, literais distintos, facilitando assim a identificação dos PCs. O exemplo abaixo ilustra esta explicação.

exemplo: utilização de comandos de controle de fluxo aninhados:

```
FAZER PARA CADA  AUX.TALIG.L.(C LOCAL='RIO').LIG (=LIGAÇÃO)
  ((  FAZER PARA CADA PC.T.(C SALARIO < 10000)  (=LIGADO)
      (( INCLUIR PC/LIGADO EM  AUX.AUMENTOS.TR
         INCLUIR PC/LIGAÇÃO.L  EM AUX.DEPTOSRIO.TR
      ))
  ))
```

Durante o processamento de um comando INÍCIO-FAZER-PARA-CADA do CI4, a PARTE I do gerador de código guar-

da a descrição do arquivo conjunto controlador, a ser criado, em uma entrada da TABPC.

Quando um comando do bloco iterativo utiliza um conjunto controlador como arquivo de entrada, sua descrição não acompanha o referido comando. Ao invés disto, o parâmetro aponta para a entrada, na TABPC, que contém esta descrição. A rotina "Declarações Globais" utiliza-se das informações desta entrada para preencher a entrada na TIR referente ao parâmetro.

A seguir é apresentada a estrutura das entradas da TABPC na forma de uma declaração de tipo de dado Pascal.

```
TYPE ENTRADA-TABPC = RECORD
                        NUMERO-CAMPOS-REG,
                        SUFIXO-ID-REG,
                        TIPO-PRIMEIRO-CAMPO,
                        TIPO-REG,     (TUPLA ou LIGAÇÃO)
                        NUMERO-CAMPOS-LIG:INTEGER
                     END;
```

As informações armazenadas na TABPC são análogas às encontradas na TIR.

A PARTE I utiliza uma ocorrência global da TABPC para tratar todos os comandos de controle de fluxo.

9.4.3.3.4 Tabela de Arquivos Intermediários (TAI)

Durante os processos de seleção de caminhos de acesso e geração de código, podem ocorrer dois tipos de situações a partir das quais devem ser gerados arquivos intermediários:

- um nodo da árvore de operações produz resultados classificados de forma não aproveitável por seu nodo su

perior. A árvore, então, deve ser dividida de modo que esta operação gere um arquivo intermediário que possa ser classificado antes de ser processado pelo nodo superior. Esta tarefa é executada pelo módulo seletor de caminhos de acesso. Este módulo pode representar arquivos temporários, no CI4, através de parâmetros do tipo arquivo e subtipos 4 ou 5;

- o comando do CI4 deve ser segmentado, pois o tamanho do código a ser gerado excede o tamanho máximo, aceitável pelo SOL, para um programa seqüencial. O comando é então subdividido em dois ou mais comandos que se comunicam através de arquivos intermediários. Esta tarefa é de responsabilidade do gerador de código.

O sistema deve, então, manter um conjunto de arquivos, em disco, disponíveis para serem utilizados nestas situações. Estes arquivos podem ser liberados após a execução dos comandos a eles associados.

A PARTE I do gerador de código mantém a descrição deste conjunto de arquivos na Tabela de Arquivos Intermediários.

As entradas da TAI possuem o seguinte formato:

```
TYPE ENTRADA-TAI = RECORD
                       AREANO,
                       FILENO,
                       ID-ARQ-INT:INTEGER
                   END;
```

Os campos AREANO e FILENO identificam o arquivo no disco [CAR 82]. O campo ID-ARQ-INT informa se este arquivo já está em uso (guarda o número lógico do arquivo intermediário que vai de 1 a 999) ou se está livre (valor-1). O número lógico de um arquivo intermediário é dado ou pelo seletor de caminhos de acesso ou pelo gerador de código.

9.4.3.4 Processamento das expressões de seleção do CI4

A rotina de declarações globais não processa parâmetros do tipo expressão de seleção. Estes parâmetros são processados durante a fase de preenchimento das rotinas pré-prontas.

As expressões de seleção do CI4 possuem características próprias que influenciaram muito o projeto do algoritmo que as manipula:

- as expressões estão escritas em notação polonesa pré-fixada que reflete as precedências de operadores da linguagem LOBAN. Uma máquina de pilha teve que ser criada para infixar estas expressões mantendo o esquema de precedências do LOBAN apesar do esquema de precedência de operadores do Pascal ser diverso;

- as expressões representam literais como referências indiretas à tabela de literais gerada pelo analisador léxico do reconhecedor e armazenada em disco. A partir destas referências o sistema pode identificar corretamente os literais na tabela. Cada referência é composta de dois valores inteiros. O primeiro identifica o bloco do arquivo onde inicia o literal e o segundo inteiro identifica o caractere do bloco, a partir do qual o literal foi gravado. Os literais, representados na tabela, apresentam dois campos: o primeiro campo indica seu tamanho em caracteres; o segundo representa o próprio literal. A figura 9.5 apresenta o esquema de referência a literais adotado pelo sistema;

- a principal característica das expressões de seleção é a possibilidade que oferecem para o uso de funções aritméticas e booleanas que também estão associadas a rotinas pré-prontas. Uma conseqüência desta flexibilidade é que durante o preenchimento de uma rotina pré-pronta, associada

a um comando, pode-se ter que preencher outras, associadas a funções aritméticas ou lógicas. Exemplo:

COLEC AUX.TALIG1.(C L.SALARIO > MEDIA SOBRE T.SALARIO)

Esta característica das expressões de seleção altera, sensivelmente, o algoritmo de preenchimento de rotinas pré-prontas.

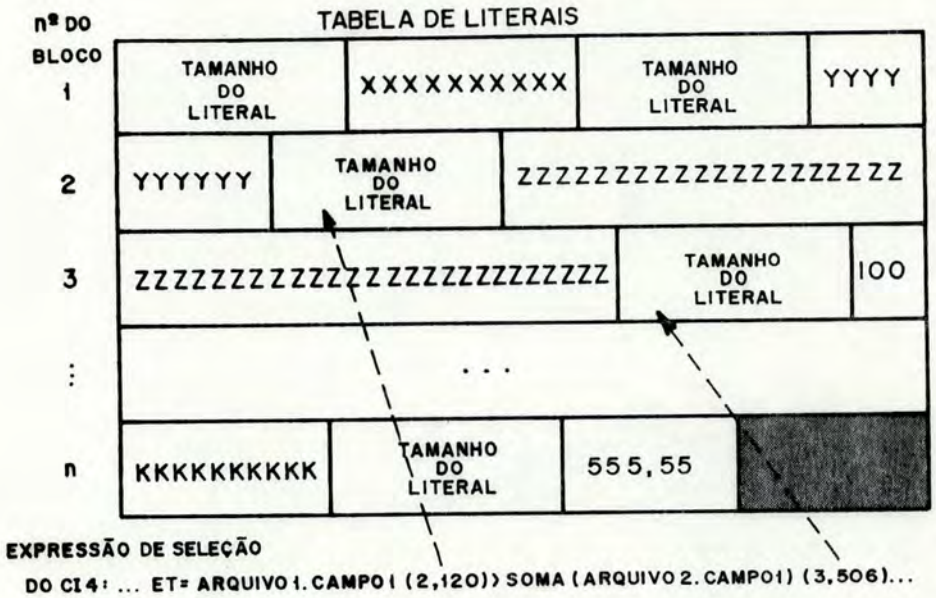

FIGURA 9.5: Referência Indireta à Tabela de Literais

A solução a este problema, adotada no projeto de implementação do módulo montador, foi processar os comandos do CI4, que apresentam expressões de seleção contendo funções, em dois passos. Quando a rotina de preenchimento de rotinas pré-prontas encontra uma referência a alguma função, durante o processamento de uma expressão de seleção, ela segue os seguintes passos:

a) gera um identificador de rotina Pascal para a função encontrada;

b) grava este identificador e a posição no CI4 onde a função aparece numa lista de rotinas a serem preenchidas;

c) grava, também, o identificador na rotina sendo preenchida como uma chamada à função a ser desenvolvida;

d) continua a processar a expressão de seleção normalmente.

Após o processamento de cada comando do CI4, a PARTE I verifica se a lista de rotinas a preencher contém identificadores. Em caso afirmativo, ela volta a processar a CI4, nos pontos indicados, criando as rotinas que implementam as funções utilizadas na expressão de seleção.

9.4.3.5 Estrutura das rotinas pré-prontas para implementação

9.4.3.5.1 Introdução

Como já foi exposto no capítulo 8, as rotinas pré-prontas processadas pelo gerador de código são compostas de trechos completos em Pascal, e de trechos a serem completados e/ou selecionados através de funções avaliadas em tempo de geração.

Dois aspectos do projeto de implementação destas rotinas mereceram maior atenção:

- a estrutura das rotinas deveria servir à implementação tanto de comandos embutidos, como de principais do CI4; enquanto os comandos principais são invocados uma só vez e geram tabelas inteiras como resultado, os comandos embutidos são controlados pelos principais a eles associados (nodos superiores na árvore de operação). A cada chamada,os comandos embutidos fornecem, ao nodo chamador, uma tupla ou ligação do resultado total;

- deveria-se resolver todos os casos de preenchimento com o menor número possível de funções.

A seguir são apresentadas a estrutura das rotinas pré-prontas, para implementação e as funções escolhidas para seu preenchimento.

9.4.3.5.2 Estrutura das rotinas pré-prontas

Em um primeiro momento pensou-se em escrever rotinas que pudessem ser preenchidas tanto para comandos externos do CI4 como para embutidos. A seleção dos trechos válidos em cada caso seria feita através de uma função. Verificou-se, mais tarde, através de alguns testes, que isto degradava em muito o processo de preenchimento das rotinas, pois a função tinha que ser avaliada em vários de seus trechos, aumentando o tempo de processamento.

Decidiu-se, então, que cada rotina pré-pronta seria representada por dois textos separados. Um para implementar seu comando associado, quando este aparecesse no CI4 como principal, e outro para implementá-lo quando aparecesse embutido. Desta forma eliminou-se uma função de alto custo em termos de tempo de processamento.

O resultado do processamento de textos, para comandos principais do CI4, são rotinas chamadas independentes, que geram arquivos intermediários como produto de sua execução.

O preenchimento de uma rotina pré-pronta para comando embutido produz três rotinas Pascal que serão chamadas pela rotina gerada para o comando mais externo.

A primeira rotina é de inicialização de variáveis globais e abertura de arquivos e/ou chamadas a rotinas que inicializam seus comandos embutidos.

A segunda rotina é, na verdade, uma função. A cada chamada, esta função devolve ao chamador uma tupla ou ligação, como resultado parcial do processamento do comando embutido.

A terceira rotina, resultante do preenchimento da rotina pré-pronta, fecha arquivos e/ou chama rotinas de finalização de comandos embutidos a este no CI4.

A seguir, é apresentado um exemplo de comando do CI4, que contém como parâmetro um comando embutido, e as rotinas Pascal geradas a partir do processamento de suas rotinas pré-prontas associadas. As rotinas estão escritas em linguagem semelhante a Pascal.

Exemplo de rotinas geradas a partir de comandos do CI4:

- comando do CI4, de forma resumida:

```
... INCLUIR (JUNTAR(ESTREITAR(A),B) em AUX.TAB) ...
```

- rotinas geradas a partir do comando ESTREITAR:

```
PROCEDURE MBOPEN<sufixo1>;
   BEGIN
      <inicializa algumas variáveis de controle,
                  um registro e um filemap para A>;
      OPEN(FILEMAP(A));
   END;

PROCEDURE MBGNEXT<sufixo1>(MBR<sufixo1>:<resultado.ESTREITAR>;
                           ESTADO:INTEGER);
   BEGIN
       READ(FILEMAP(A));
       IF FILEMAP(A).RESULT <> ENDOFILE
       THEN  BEGIN
             <estreita registro de A em MBR<sufixo1> >;
             ESTADO:= OK
             END
       ELSE ESTADO:= ENDOFILE;
   END;

PROCEDURE MBCLOSE<sufixo1>;
   BEGIN
       <ajusta alguns atributos do filemap de A>;
       CLOSE(FILEMAP(A));
   END;
```

- rotinas geradas a partir do comando JUNTAR:

```
PROCEDURE MBOPEN<sufixo2>;
   BEGIN
       <inicializa algumas variáveis de controle,
                   um registro e um filemap para B>;
       MBOPEN<sufixo1>: "chama a rotina de inicialização do
                         comando ESTREITAR"
       OPEN(FILEMAP(B));
   END;

PROCEDURE MBGNEXT<sufixo2>(MBR<sufixo2>:<resultado.JUNTAR>;
                           ESTADO:INTEGER);
   BEGIN
       MBGNEXT<sufixo1>(MBR<sufixo1>, J-ESTADO);
       IF J-ESTADO <> ENDOFILE
       THEN BEGIN
              READ(FILEMAP(B));
              <obtém uma tupla do resultado em MBR<sufixo2>>;
              J-ESTADO:= OK
            END
       ELSE J-ESTADO:= ENDOFILE;
   END;

PROCEDURE MBCLOSE<sufixo2>;
   BEGIN
       <ajusta alguns atributos do filemap de B>
       CLOSE(FILEMAP(B));
       MBCLOSE<sufixo1>; "finaliza o comando ESTREITAR"
   END;
```

- rotina gerada a partir do comando INCLUIR:

```
PROCEDURE MBPROC<sufixo3>;
   BEGIN
       <cria,no Heap, suas áreas globais e as áreas que
        serão utilizadas por seus comandos embutidos>;
       MBOPEN<sufixo2>; "chama a rotina de inicialização do
                         comando JUNTAR"
       OPEN(FILEMAP(C));
       MBGNEXT<sufixo2>(MBR<sufixo2>; C-ESTADO);
       WHILE C-ESTADO = OK
       DO BEGIN
             WRITE(FILEMAP(C),MBR<sufixo2>); "realiza inclu
                                              são em C"
             MBGNEXT sufixo2 (MBR<sufixo2>, C-ESTADO)
          END;
       MBCLOSE<sufixo2>; "finaliza o comando JUNTAR"
       CLOSE(FILEMAP(C));
       IF C-ESTADO=ENDOFILE  "atualiza a variável de estado"
       THEN MBESTADO:= OK    "do sistema"
       ELSE MBESTADO:= C-ESTADO;
   END;
```

9.4.3.5.3 Funções de controle para as rotinas pré-prontas

Abaixo são apresentadas as funções de controle das rotinas pré-prontas da biblioteca do gerador de código do Sistema L.

a) MOVE1:

parâmetros:

- índice, na TIR, do registro fonte;
- índice, na TIR, do registro destino;

semântica:

escreve trecho em Pascal atribuindo os valores dos campos do registro fonte aos campos do registro destino. Os registros devem ser do mesmo tipo.

b) MOVE2:

parâmetros:

- índice, na TIR, do registro fonte;
- número do campo do registro fonte;
- índice, na TIR, do registro destino;
- número do campo do registro destino;

semântica:

atribui a um campo do registro destino o valor de um campo do registro fonte.

c) MOVE3:

parâmetro:

- índice, na TIR, da LDA;

semântica:

preenche os campos da LDA com valores obtidos de seus campos de origem.

d) COMPARA1:

parâmetros:

- índice, na TIR, da primeira LDA;
- índice, na TIR, da segunda LDA;
- operador relacional (= ou <>);
- identificador de variável booleana local;

semântica:

avalia a expressão booleana formada por LDA1, operador relacional e LDA2; atribui, à variável booleana, o resultado da expressão.

e) COMPARA3:

parâmetros:

- índice, na TIR, da primeira LDA;
- índice, na TIR, da segunda LDA;
- operador relacional (<,>);
- identificador de variável booleana local;

semântica:

igual a de COMPARA1.

f) TRATA-EXPRESSÃO-DE-SELEÇÃO:

parâmetros:

não tem;

semântica:

processa a expressão de seleção (que se encontra no CI4) da forma explicada na seção 9.4.3.3.

g) INSERE-SUFIXO:

parâmetro:

- índice, na TIR, do arquivo ou LDA;

semântica:

substitui a chamada da função pelo valor do campo

SUFIXO-ID-REG da entrada da TIR.

h) INICIALIZA-REGISTRO:

parâmetro:

- índice, na TIR, do arquivo ou LDA;

semântica:

função utilizada para inicializar registros; verifica, na TIR, o tipo do primeiro campo do registro e atribui a ele um valor convencionado como valor inicial.

i) INSERE-MARK-NO:

parâmetros:

- indicador da operação (O-MARK; 1-RELEASE);

semântica:

cria, a partir de um contador global, um identificador de MARK (marca) do Heap. Estes identificadores são usados por primitivas do SOL que alocam e liberam áreas da memória dinâmica [CAR 82].

j) TRATA-VAR-LOCAL:

parâmetros:

não tem;

semântica:

insere sufixo no registro de variáveis locais a partir da variável CONTLOCAIS.

k) SEL-TIPOARQ:

parâmetro:

- índice, na TIR, do arquivo;

semântica:

seleciona trechos alternativos, que seguem,dependendo do valor do campo TIPO-ARQUIVO da entrada da TIR.

l) DECLARA-LOCAIS:

parâmetros:

não tem;

semântica:

declara as variáveis locais codificadas, a seguir, no próprio texto da rotina sendo processada.

m) INICIALIZA-FILEMAPS:

parâmetros:

não tem;

semântica:

a partir dos valores dos campos AREANO e FILENO das entradas da TIR e, usando os sufixos também armazenados na tabela, inicializa os descritores dos arquivos que serão utilizados pela rotina.

n) INSERE-SUFIXO-PROC:

parâmetros:

não tem;

semântica:

monta, a partir de um contador global, um sufixo para o identificador de rotina Pascal sendo gerado.

o) SEL-TIPOPARAM:

parâmetros:

- número do parâmetro do comando,no CI4(1 ou 2);

semântica:

seleciona trechos alternativos da rotina a partir de uma variável booleana, atualizada durante o processamento do comando no CI4, que indica se o parâmetro é do tipo comando ou não.

p) INSERE-SUFIXO-FILHO:

parâmetros:

- indicador do tipo de sufixo (1- da rotina; 2-do registro);
- número do parâmetro do comando, no CI4(1 ou 2);

semântica:

monta, no texto, o sufixo do identificador de uma rotina de comando embutido a ser chamada, ou o sufixo do registro passado como parâmetro na chamada de uma rotina do tipo MBGNEXT.

q) GUARDA-INFO-FILHO:

parâmetro:

- número do parâmetro do comando,no CI4(1 ou 2);

semântica:

armazena em duas variáveis o sufixo das rotinas do comando embutido e o sufixo do registro, passado como parâmetro, na sua rotina MBGNEXT.

r) SUFIXO-PONTEIRO-GLOBAL:

parâmetros:

não tem;

semântica:

insere, no texto, um número de 3 dígitos correspondente ao valor de um contador de ponteiros globais.

9.4.4 Algoritmo de funcionamento do módulo intercalador (PARTE II)

A seguir é apresentado o algoritmo de funcionamento do módulo intercalador (PARTE II) do gerador de código.

- Algoritmo de funcionamento da PARTE II:

```
1 - TIPO-DO-GRUPO:= NÃO-PRINCIPAL;
2 - ENQUANTO TIPO-DO-GRUPO = NÃO-PRINCIPAL
    FAÇA:
    2.1 - BUSCA GRUPO DE ARQUIVOS TEMPORÁRIOS PARA INTERCALAÇÃO (MONITOR);
    2.2 - SE TIPO-DO-GRUPO = NÃO-PRINCIPAL
          ENTÃO FAÇA:
                - GRAVA NO ARQUIVO DESTINADO AO PROGRAMA
                  PASCAL, NESTA ORDEM, OS ARQUIVOS TEMPORÁRIOS:TIPOS DE DADOS, VARIÁVEIS,DECLARAÇÕES
                  DE ROTINAS E CHAMADAS;
                FIM-FAÇA
    2.3 - SENÃO "TIPO-DO-GRUPO = PROGRAMA-PRINCIPAL"
                FAÇA:
              2.3.1 - EXECUTA A ROTINA "INICIALIZA-VARIÁVEIS";
              2.3.2 - ENQUANTO VAR-DECLARAÇÕES
                      FAÇA:
                  2.3.2.1 - LÊ CARACTERES DO PROGRAMA PLOBAN E GRAVA NO ARQUIVO PASCAL ATÉ ENCONTRAR UM
                            DOS SEGUINTES PADRÕES: VAR, PROCEDURE,
                            ## (na primeira coluna);
                  2.3.2.2 - CASO O PADRÃO ENCONTRADO
                            FOR:
                                VAR:
                                   - EXECUTA A ROTINA
                                     "TYPE-VAR-DCL";
```

PROCEDURE:
- SE VAR-BOOL
  ENTÃO EXECUTA A ROTINA
  "TYPE-VAR-DECL";
- EXECUTA A ROTINA
  "PROC-DCL";

##

- SE VAR-BOOL
  ENTÃO EXECUTA A ROTINA
  "TYPE-VAR-DECL";
- SE PROC-BOOL
  ENTÃO EXECUTA A ROTINA
  "PROC-DCL";
- VAR-DECLARAÇÕES:=FALSO;

FIM-FAÇA;

2.3.3 - LÊ O PROGRAMA PLOBAN ATÉ ENCONTRAR UMA LINHA DE CÓDIGO SEM O CARACTERE ## NA COLUNA 1;

2.3.4 - ATÉ O FIM DO ARQUIVO DE CHAMADAS
FAÇA:
- LÊ O PRÓXIMO CARACTERE DO ARQUIVO DE CHAMADAS
- SE CARACTERE <> MARCA-PASCAL
  ENTÃO GRAVA O CARACTERE NO PROGRAMA PASCAL
- SENÃO ATÉ CARACTERE = ## OU FIM-ARQUIVO-PLOBAM

FAÇA:
- LÊ PRÓXIMO CARACTERE DO PROGRAMA PLOBAM
- GRAVA O CARACTERE NO PROGRAMA PASCAL;

FIM-FAÇA;

FIM-FAÇA;

```
    2.4 - LIBERA GRUPO DE ARQUIVOS TEMPORÁRIOS PARA PARTE I
          (MONITOR);
    2.5 - INVOCA A PARTE III PARA EXECUÇÃO USANDO O ARQUIVO
          PASCAL;
FIM-FAÇA;
```

- *Inicializa Variáveis:*

```
1 - VAR-DECLARAÇÃO := VERDADEIRO;
2 - VAR-BOOL := VERDADEIRO;
3 - PROC-BOOL := VERDADEIRO;
```

- *Type-Var-Dcl:*

```
1 - GRAVA 'TYPE' NO PROGRAMA PASCAL;
2 - GRAVA O CONTEÚDO DO ARQUIVO DE TIPOS NO PROGRAMA PASCAL;
3 - GRAVA 'VAR' NO PROGRAMA PASCAL;
4 - GRAVA O CONTEÚDO DO ARQUIVO DE VARIÁVEIS NO  PROGRAMA
    PASCAL;
5 - VAR-BOOL := FALSO;
```

- *Proc Dcl:*

```
  - GRAVA O CONTEÚDO DO ARQUIVO DE DECLARAÇÕES DE ROTINAS
    NO PROGRAMA PASCAL;
  - PROC-BOOL := FALSO;
```

9.4.5 Algoritmo de funcionamento do módulo compilador (PARTE III)

A seguir, é apresentado o algoritmo de funcionamento do módulo de compilação (PARTE III) do gerador de código.

- *Algoritmo de funcionamento da PARTE III:*

1 - INVOCA O COMPILADOR PASCAL USANDO A IDENTIFICAÇÃO DO ARQUIVO PASCAL GERADO PELA PARTE II;

2 - SE O ARQUIVO PASCAL É O PROGRAMA PRINCIPAL
ENTÃO FAÇA:
- ABRE O ARQUIVO OBJETO;
- GRAVA NO ARQUIVO OBJETO, A PARTIR DO FIM DO CÓDIGO GERADO, DATA E HORA ATUAIS, LISTA DE ESTRUTURAS DO CI4 E O CI2 COMPLETO;
- FECHA O ARQUIVO OBJETO

FIM FAÇA;

9.4.6 Variável de estado da Interface LOBAN Embutida

A Interface LOBAN Embutida oferece ao usuário PLOBAN uma variável de estado (MBESTADO), que pode ser consultada a qualquer momento e que informa o tipo de terminação da última instrução LOBAN executada.

Atualmente, esta variável pode assumir os valores FALSE ou TRUE, que representam, respectivamente, término normal ou anormal da última instrução LOBAN processada.

A variável MBESTADO é declarada, automaticamente, no programa Pascal principal criado pelo gerador de código.

9.4.7 Limitações ao usuário PLOBAN

A seguir é apresentado o conjunto de restrições e exigências ao usuário causadas por este projeto de implementação do módulo gerador de código da interface embutida do Sistema L:

a) usuário não pode declarar identificadores Pascal que iniciem pelos caracteres MB;

b) as funções MARK e RELEASE não devem ser usadas dentro dos blocos iterativos das instruções LOBAN de controle de fluxo (FAZER PARA CADA);

c) todas as variáveis Pascal que serão utilizadas em instruções LOBAN REPRESENTAR ou pelo operador INT (interpretar), de expressões de obtenção de construções, devem ser declaradas no Heap do programa;

d) os administradores de base de dados não podem criar atributos do tipo CONJUNTO DE CARACTERES com mais de 998 elementos;

e) a descrição de tuplas não pode conter mais que 255 atributos;

f) as tuplas resultantes da execução de operadores de expressões de obtenção de construções não podem ser de tamanho maior que 512 caracteres;

g) atributos do tipo CONJUNTO DE CARACTERES não podem ter número ímpar de elementos; esta restrição é um reflexo de restrição análoga, apresentada pelo compilador Pascal;

h) as funções de expressões de seleção dos tipos ELEM, CONT, CARD, SOMA, MEDIA, DESV, MAX e MIN não podem aparecer embutidas umas nas outras.

9.4.8 Modificações no ambiente para a implementação do gerador

9.4.8.1 Introdução

Como já foi colocado, no início deste capítulo, tentou-se manter um compromisso com a performance do gerador. Neste sentido decidiu-se, não só adaptar o projeto às condições apresentadas pelo ambiente como, também, alterar algumas características do meio para facilitar o seu processamento.

A seguir, são apresentadas as mudanças, no meio ambiente, propostas pelo projeto de implementação do módulo gerador de código.

9.4.8.2 Alterações propostas para o SOL

Para aumentar a eficiência do gerador e possibilitar a implementação assíncrona de suas partes, são propostas alterações de ordem estrutural no SOL além de novos serviços que este sistema deve oferecer.

O SOL é um programa em Pascal Concorrente composto de vários processos [CAR 82], como mostra a figura 9.6.

Os processos de programas Pascal Concorrentes dispõem, cada um, de uma área de tamanho fixo para a execução de programas em Pascal Seqüencial. No caso do SOL, o tamanho desta área, em cada processo, foi decidido levando-se em consideração o tamanho do maior programa seqüencial que por ele seria ativado. Além disso, o tamanho total do sistema não poderia exceder os 96 Kbytes de memória principal oferecidos pelo LABO 8034 do CPGCC-UFRGS.

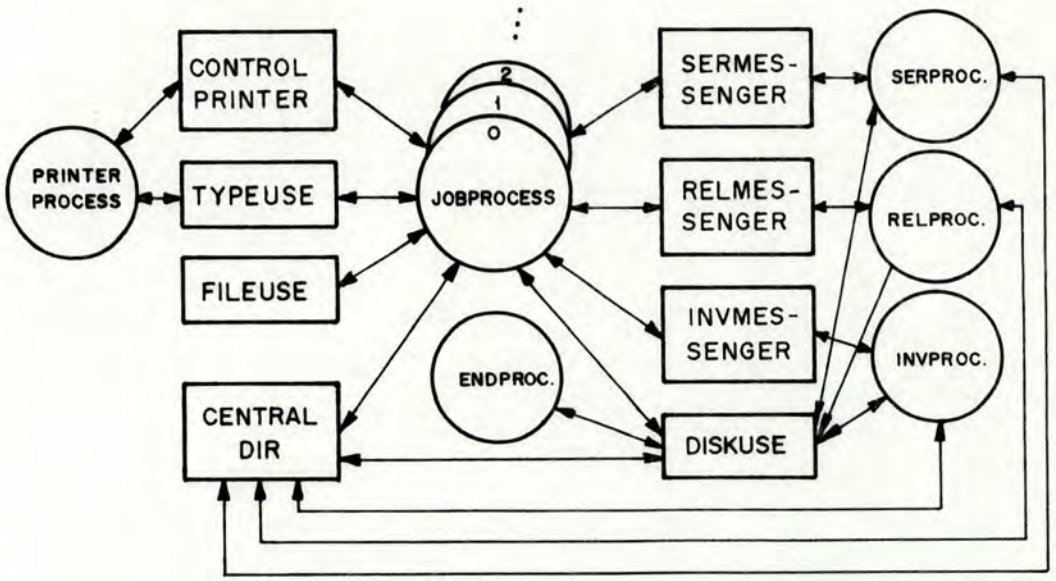

FIGURA 9.6: Arquitetura do Programa SOL

Os processos do SOL podem ser divididos, basicamente, em três categorias distintas: 1) os que gerenciam os recursos periféricos da máquina (ex.: CONTROLPRINTER que gerencia o uso da impressora); 2) os que implementam o método de acesso (ex.: INVPROCESS que é responsável pelos arquivos de inversão) e 3) os que ativam programas de usuários (processos do tipo JOBPROCESS).

Os tamanhos das áreas de programas seqüenciais, dos processos de gerência de recursos e de implementação do método de acesso, não precisam ser grandes, pois elas mantêm programas específicos e pequenos. Já os processos de usuários (do tipo JOBPROCESS) devem dispor de áreas maiores, pois não se sabe a priori os tamanhos dos programas que serão por eles ativados.

Devido ao tamanho restrito de memória oferecido pelo hardware, a versão atual do SOL foi gerada com, somente, um JOBPROCESS. Portanto, do ponto de vista do usuário,o sistema é monoprogramável.

A tabela 9.1, apresentada abaixo, associa cada elemento do SOL a seu tamanho em bytes.

ELEMENTO	ESPAÇO OCUPADO (EM BYTES)
Kernel/Interpretador	10240
Variáveis e stack do SO	5998
Cod. Obj. do Programa SOL	12288
Área de Progr.JOBPROCESS	18432
Área de dados JOBPROCESS	14000
Área de Progr.SERPROCESS	3584
Área de dados SERPROCESS	3700
Área de Progr. RELPROCESS	5120
Área de dados RELPROCESS	4300
Área de Progr. INVPROCESS	6144
Área de dados INVPROCESS	6400
TOTAL	90206

TABELA 9.1: Tamanhos dos Elementos do Sistema SOL

Para implementar o módulo gerador de forma assíncrona, a PARTE I deve executar em um processo enquanto as partes II e III são ativadas a partir de outro.

O compilador Pascal, invocado pela PARTE III, ocupa uma área de 18 Kbytes. Sendo assim, decidiu-se que as partes II e III seriam ativadas pelo JOBPROCESS.

Logo ficou claro, que as áreas de programa, oferecidas pelos outros processos do sistema seriam, em separado, muito pequenas para abrigar a PARTE I do gerador. A opção de segmentar o código desta parte em vários programas de, no máximo 6 Kbytes não é realista, pois o tempo gasto na troca de programas, em tempo de execução, seria intolerável.

Propôs-se, então, a unificação dos processos INVPROCESS e RELPROCESS com uma possível diminuição no núme

ro de buffers de cada um. Com isto, obtém-se uma área de programas de, aproximadamente, 12 Kbytes. A segmentação da PARTE I, neste caso, seria reduzida à metade.

A junção destes dois processos em um, também facilitará a implementação de controle de transações e "locking" no Sistema L, pois a gerência de todos os arquivos da base de dados ficaria sob a responsabilidade de um só processo.

É possível que com esta decisão, a gerência de buffers do sistema fique mais eficiente também. Teria-se um gerente de buffers para toda a base de dados do sistema.

Para implementar a comunicação entre as partes I e II do gerador é preciso criar um novo monitor no sistema. Este monitor já foi referenciado pelos algoritmos de funcionamento destas partes e é apresentado, a seguir, em linguagem semelhante a Pascal Concorrente.

```
TYPE ARQ-TEMPORÁRIOS = MONITOR;

    CONST  BASE = Ø;
           TOTGRUPOS = n;
           FILENO1 = k;
    TYPE   TIPO-GRUPO = (COMUM, PRINCIPAL);
           GRUPOS = ARRAY (.1..n.) OF RECORD NRO:INTEGER;
                    TIPO:TIPO-GRUPO END;
    VAR    GRUPO-VAZIO, GRUPO-CHEIO:GRUPOS;
           TOPO-V, TOPO-C:INTEGER;
           ESPERA-V, ESPERA-C:QUEUE;

  PROCEDURE ENTRY BUSCA-GRUPO-VAZIO(VAR NUM-GRUPO:INTEGER);
     BEGIN
           IF  TOPO-V = BASE
           THEN DELAY(ESPERA-V);
           NUM-GRUPO:= GRUPO-VAZIO(.TOPO-V.).NRO;
           TOPO-V:= TOPO-V - 1;
     END;
```

```
PROCEDURE ENTRY LIBERA-GRUPO-CHEIO (TIPO-G:TIPO-GRUPO;
                                    NUM-GRUPO:INTEGER);
   BEGIN
        TOPO-C := TOPO-C + 1;
        WITH GRUPO-CHEIO(.TOPO-C.)
        DO BEGIN
             NRO:= NUM-GRUPO;
             TIPO:= TIPO-G
           END;
        CONTINUE(ESPERA-C);
   END;

PROCEDURE ENTRY BUSCA-GRUPO-CHEIO(VAR TIPO-G:TIPO-GRUPO;
                                  NUM-GRUPO:INTEGER);
   BEGIN
        IF TOPO-C = BASE
        THEN DELAY(ESPERA-C);
        WITH GRUPO-CHEIO(.TOPO.C.)
        DO BEGIN
             TIPO-G:= TIPO;
             NUM-GRUPO:= NRO
           END;
   END;

PROCEDURE ENTRY LIBERA-GRUPO-VAZIO(NUM-GRUPO:INTEGER);
   BEGIN
        TOPO-V:= TOPO-V + 1;
        GRUPO-V(.TOPO-V.).NRO := NUM-GRUPO;
        CONTINUE(ESPERA-V);
   END;

BEGIN          "INICIALIZAÇÃO DO MONITOR"
        TOPO-C := Ø;
        FOR TOPO-V := 1 TO TOTGRUPOS
        DO  GRUPO-VAZIO(.TOPO-V).NRO := 4*(TOPO-V - 1)+FILENO1;
        TOPO-V := TOTGRUPOS;
END;
```

A última alteração proposta para o SOL é a implementação de um sistema de diretório que associe as informações de data de criação e nome externo, entre outras, aos identificadores internos dos arquivos (AREANO e FILENO). Esta alteração não só facilitará a comunicação entre o usuário e o sistema, como seria um passo para o transporte de todos os utilitários e do compilador Pascal que executam no Sistema DUO [CAR 81], da mesma instalação, para o SOL.

10. AVALIAÇÃO E CONCLUSÕES

Foi proposta uma nova arquitetura para o Sistema L, que implementa um subconjunto de LOBAN. Esta nova arquitetura apresenta como característica principal, que a diferencia da antiga, o oferecimento de duas interfaces para a comunicação com o usuário, ao invés de uma. Uma interface autônoma para administração da base de dados, que interpreta as instruções do usuário (implementada por programas que correspondem aos interpretadores especializados da interface única da arquitetura antiga), e uma interface embutida em Pascal (programas PLOBAN) para a manipulação dos dados da base de dados interna. A interface embutida segue um esquema compilativo, também proposto neste trabalho, e é implementada por um pré-processador de programas PLOBAN.

Foi apresentada uma técnica de geração de código, para esquemas compilativos de linguagens de banco de dados não-procedurais, e o projeto de implementação do módulo gerador de código, da interface embutida do Sistema L, baseado nesta técnica.

O módulo gerador de código utiliza-se de uma biblioteca de rotinas pré-prontas que, quando preenchidas, são combinadas para formar programas em Pascal, funcionalmente iguais ao programa PLOBAN do usuário. Estes programas são então submetidos ao compilador Pascal, já existente na instalação que, a partir deles, gera código P executável.

A seguir, são apresentados alguns itens de avaliação sobre o trabalho desenvolvido:

a) os sistemas que adotam esquemas compilativos que geram, em tempo de pré-processamento, código completo a partir das consultas do usuário, não podem garantir que a tarefa de seleção de caminhos de acesso (também executada

neste tempo) seja feita de forma eficiente. Isto acontece pelo fato de o sistema não dispor, em tempo de pré-processamento, de informações precisas sobre o ambiente a ser encontrado, em tempo de execução das consultas. Acredita-se porém, que o possível acréscimo no tempo de execução, conseqüente da não seleção da melhor estratégia de acesso pode ser em parte compensado, pela diminuição do mesmo tempo, decorrente da transferência desta tarefa de seleção (que pode ser bastante complexa) para a fase de pré-processamento;

b) acredita-se que a tarefa de segmentação de comandos do CI4, que geram programas maiores que os permitidos, poderia ser feita de maneira mais eficiente, se fosse considerado o tipo de operação de cada comando embutido (no do filho na árvore de operações), tentando-se separar aquele que, possivelmente, irá gerar o menor arquivo intermediário; para se ter certeza disso, é necessário observar o comportamento do sistema, em atividade, e verificar-se a freqüência com que ocorrem as operações de segmentação de comandos;

c) uma medida que visa aumentar a eficiência do gerador de código, mas que não foi tomada para não aumentar a complexidade do analisador léxico, refere-se a uma avaliação precisa, que seria feita por este módulo, do número de identificadores declarados no programa PLOBAN. Este valor seria então utilizado nos cálculos de tamanho-atual feito pelos algoritmos da PARTE I do gerador de código. Atualmente, este cálculo baseia-se em estatísticas que fornecem o número médio de identificadores declarados em programas PLOBAN;

d) a decisão de dotar a nova arquitetura proposta para o Sistema L de uma interface embutida, para a manipulação da base de dados, facilitou o projeto de implementação do sistema. O usuário deverá realizar operações de entrada e saída e processamento de expressões através de instruções Pascal.

É possível salientar-se as seguintes vantagens a respeito da técnica de geração de código criada:

a) facilidade de ampliação do conjunto de instruções não-procedurais, da linguagem fonte, pela simples alteração e/ou criação de novas rotinas pré-prontas;

b) a portabilidade e a manutenção do gerador ficam facilitadas, já que tanto ele como as rotinas pré-prontas são completamente escritos em linguagem de alto nível;

c) portabilidade do código gerado que pode ser executado por qualquer implementação de máquina P;

d) facilidade de otimização do código a gerar. Cada instrução não-procedural pode dispor de mais de uma rotina pré-pronta a ela associada. A seleção da rotina mais apropriada, a cada ocorrência da instrução, pode ser feita pelo seletor de caminhos de acesso do sistema, de acordo com as estruturas de dados disponíveis na base de dados;

e) facilidade e eficiência na segmentação dos programas gerados. Este esquema automatizado de segmentação de programas representa maior eficiência para sistemas implementados sobre máquinas de pequeno porte. O usuário fica livre da tarefa de segmentar seu código e o sistema utiliza algoritmos que procuram segmentar os programas de forma eficiente;

f) diminuição no esforço de desenvolvimento do gerador de código pela utilização de compiladores, já existentes, para linguagens mais simples.

A principal desvantagem apresentada por esta técnica, em relação à geração direta de código executável, é o tempo de processamento. A execução da PARTE I do gerador é feita de forma interpretativa e relativamente lenta. Outro fator que contribui para o aumento no tempo de execução do gerador é a geração de código intermediário de alto nível.

Além de submeter as instruções do usuário aos processos de reconhecimento, análise semântica e geração de código, processos análogos devem ser ativados para as instruções geradas, a fim de se obter código executável.

ANEXO 1 SINTAXE COMPLETA DE UM ENDEREÇO DE PONTO

```
<endereço de ponto> ::= {<id ponto>|
                        [<id ponto>.]<critério seleção-
                                     ponto imediato>...}
<id ponto>::= ACTRAB|AUX|OCOR|PC[/<lit marca>]|COBTIDA
<critério seleção ponto imediato>::=(<obter valor booleano>)|
                                    <lit nome>|
                                    <d tipo>|<d prétipo>
<obter valor booleano>::= (<obter valor booleano>)|
                          NÃO <obter valor booleano>|
                          <obter valor booleano>{ET|OU|IMPL}
                          <obter valor booleano>|
                          <termo valbool>
<termo valbool> ::= VERDADEIRO |
      <expressão de obtenção de construções>{=|<>|<|>|<=|=>}
                  <expressão de obtenção de construções>|
      <expressão de obtenção de construções> ELEM
                 {<obter coleção>|<d tipo>|<d prétipo>}
<obter coleção>::= <expressão de obtenção de construção-
                    tipo  tabela relacional>|
                    <expressão de obtenção de construção-
                    tipo tabela ligacional>|
                    <expressão de obtenção de construção-
                    tipo coleção de itens>
```

ANEXO 2 EVOLUÇÃO DA ARQUITETURA DO GERADOR DE CÓDIGO

Durante o projeto de implantação do gerador de código, sua arquitetura foi sofrendo modificações. As causas destas modificações foram várias, podendo-se citar restrições apresentadas pelo ambiente de implementação, desconhecimento de detalhes de funcionamento de outros módulos e tentativas de melhorar a eficiência do módulo gerador de código.

A figura A2.1 apresenta a primeira arquitetura projetada para o gerador. Ela apresenta duas características principais:

a) foi projetada para ser chamada como uma rotina do seletor de caminhos de acesso que processa, a cada chamada, um comando do CI4;

b) ainda não se tinha conhecimento de limitações do ambiente como tamanho máximo de programas seqüenciais e a ordem rígida das declarações em programas Pascal.

O segundo esboço de arquitetura (figura A2.2) já considerava o gerador de código como um módulo à parte que processa o CI4 de todo um programa PLOBAN a cada chamada. Ainda não havia se sentido, porém, a necessidade de segmentar o programa Pascal gerado.

A figura A2.3 apresenta a terceira versão projetada da arquitetura do módulo gerador de código para o Sistema L. Ela apresenta 4 características principais que a diferenciam das anteriores:

a) aventou-se a possibilidade de gerar programas seqüenciais se o tamanho do código gerado ficasse muito grande. Só não se havia percebido que mesmo um só comando do CI4 pode gerar declarações em alguma ordem, que não a aceita pelo compilador Pascal;

b) a leitura de um arquivo que marca a posição, no texto, dos trechos em Pascal do programa PLOBAN (marcas-Pascal). Este arquivo auxilia o processamento da PARTE II para inserir os trechos Pascal do programa PLOBAN durante a intercalação para gerar o programa principal em Pascal;

c) o grupo de arquivos temporários do programa principal é ampliado pela criação de um arquivo com declarações "FORWARD" das rotinas de controle de fluxo que podem ser chamadas por rotinas do usuário e chamar rotinas deste. As rotinas geradas seriam declaradas após as do usuário.

A quarta versão da arquitetura do gerador reflete a necessidade de se gerar programas seqüenciais secundários da mesma forma que o programa Pascal principal (figura A2.4).

A arquitetura atual do gerador de código para implementação está representada no capítulo 9, através da figura 9.2 e apresenta, como fato novo, a supressão do arquivo de declarações "FORWARD". Este arquivo tornou-se dispensável a partir da decisão, de projeto, de gerar código para os comandos FAZER PARA CADA diretamente no arquivo de chamadas do programa principal (como parte de seu bloco externo), e não mais como rotinas a serem chamadas.

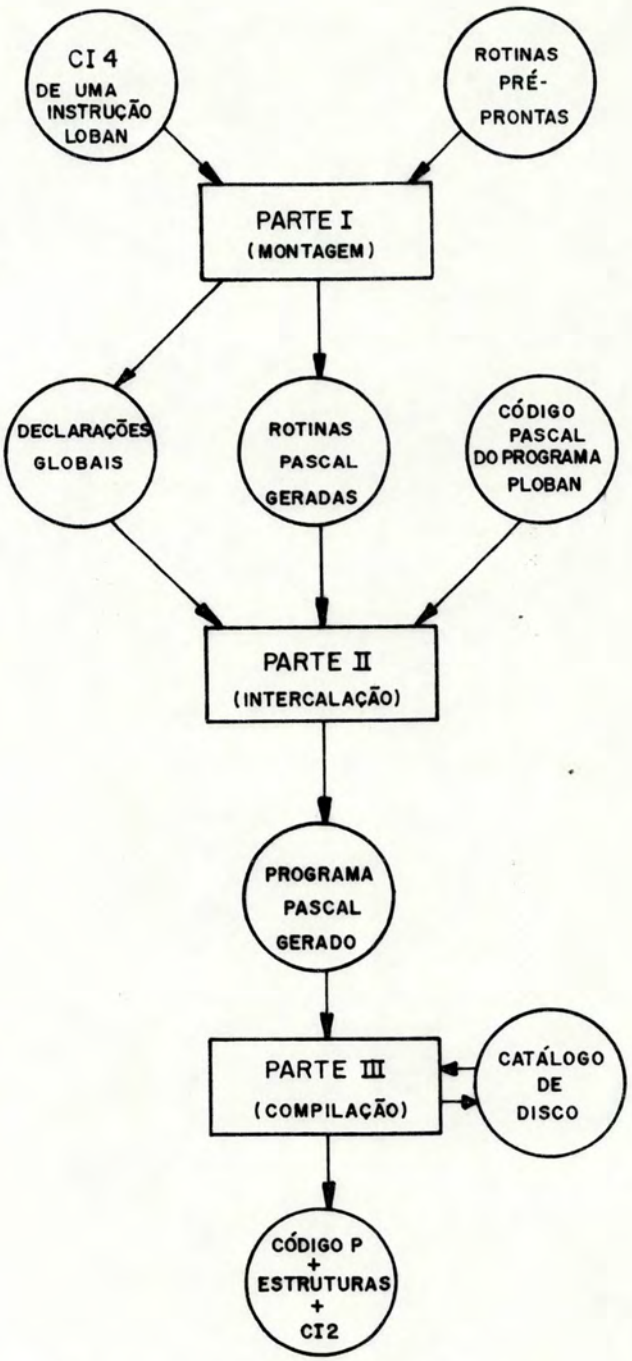

FIGURA A2.1: 1ª Arquitetura do Módulo Gerador de Código

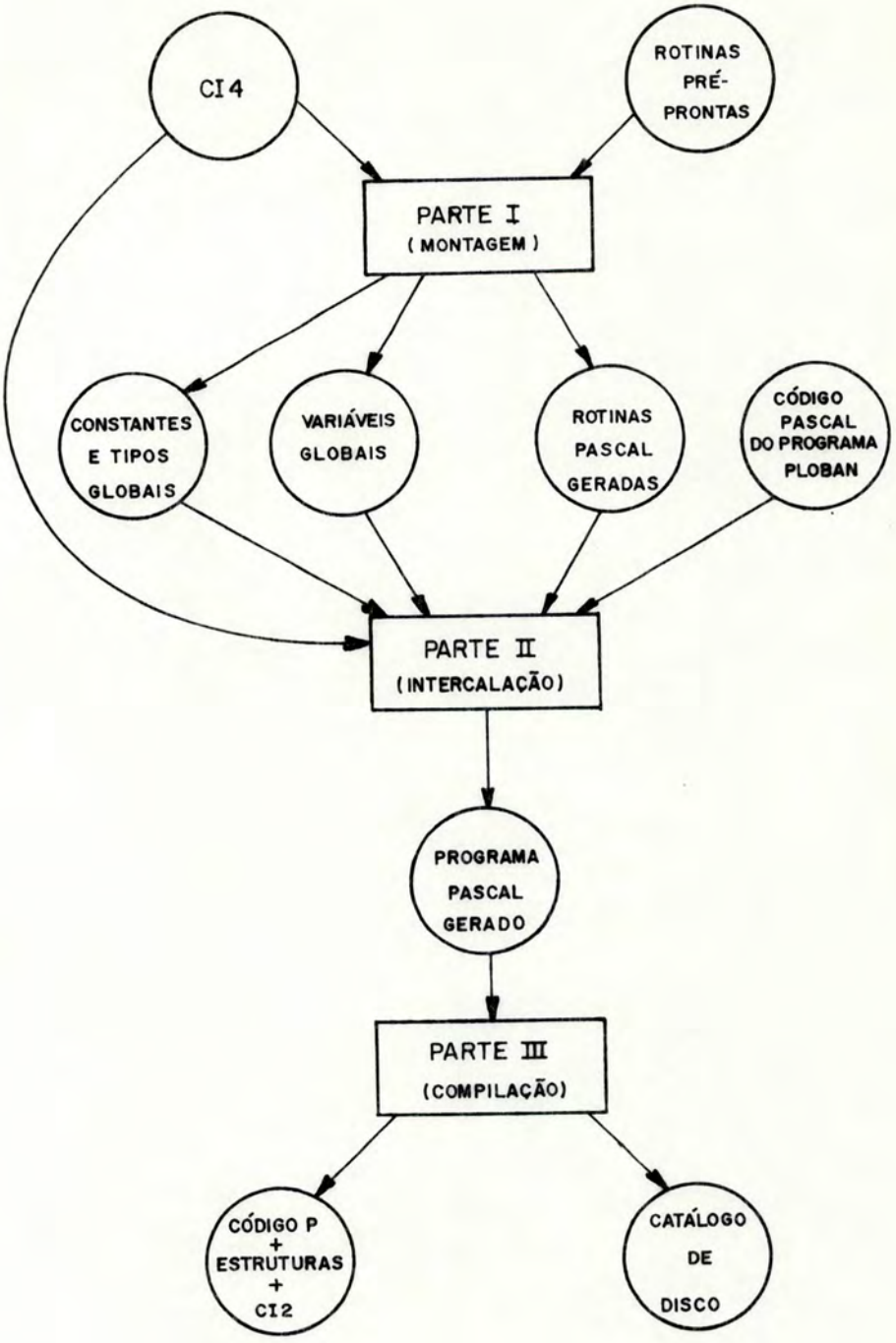

FIGURA A2.2: 2ª Arquitetura do Módulo Gerador de Código

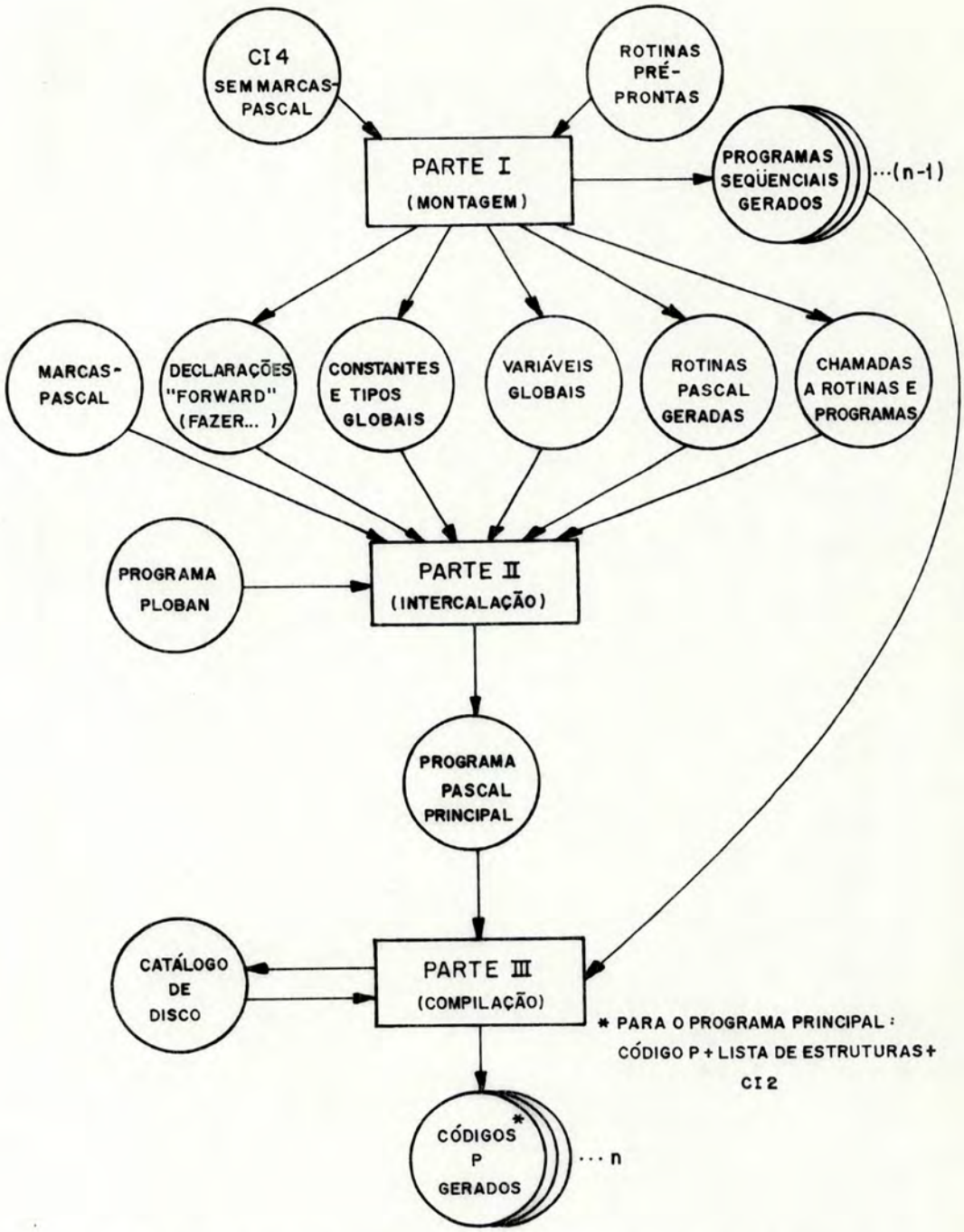

FIGURA A2.3: 3ª Arquitetura do Módulo Gerador de Código

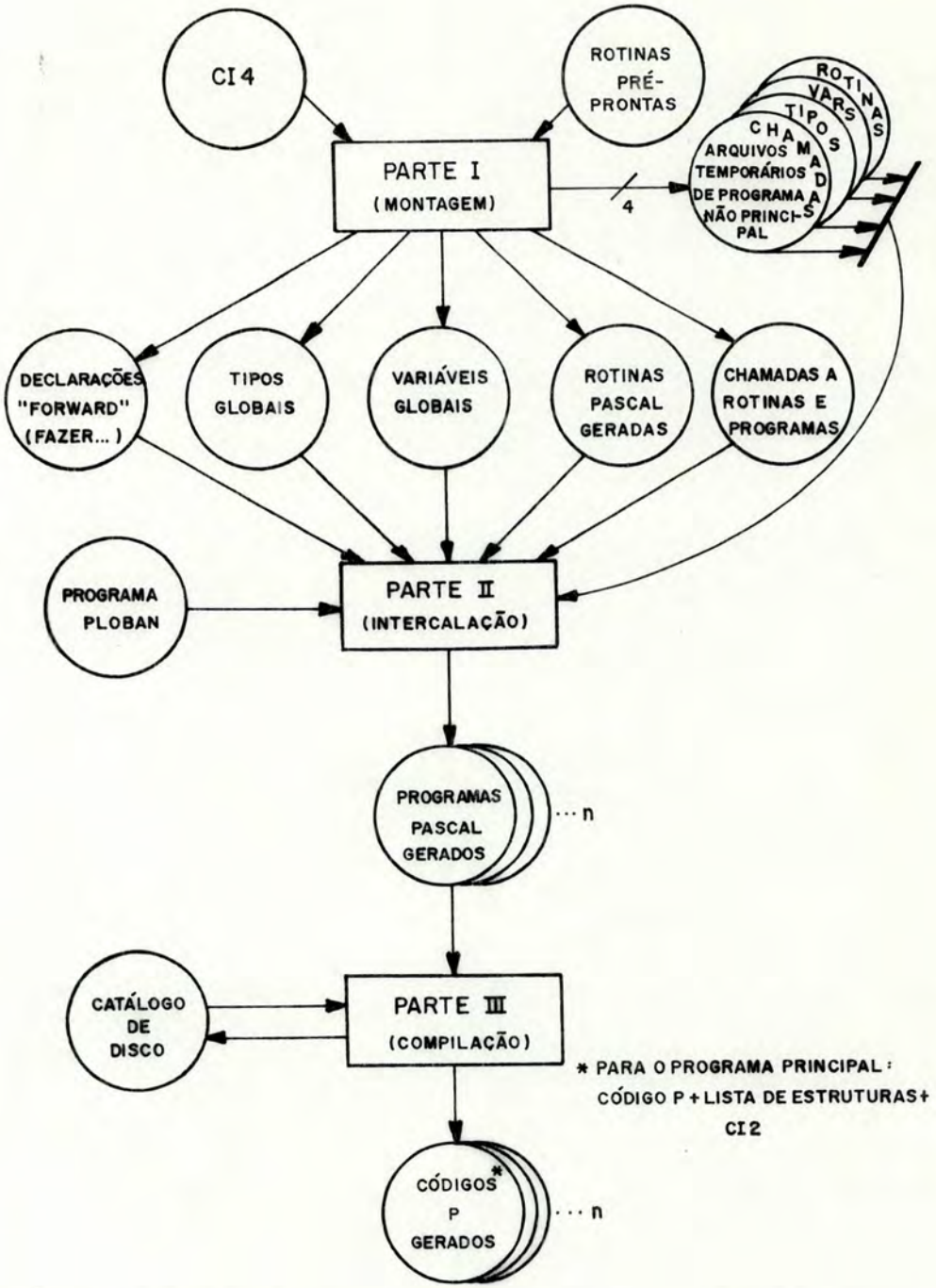

FIGURA A2.4: 4ª Arquitetura do Módulo Gerador de Código

ANEXO 3 SINTAXE COMPLETA DO CÓDIGO INTERMEDIÁRIO 4 (CI4)

```
<CI4>::= <id-prog-PLOBAN>(<marca-Pascal>|<comando>...)
                          <lista-de-estruturas>
    <id-prog-PLOBAN>::=<AREANO><FILENO>
        <AREANO>::= INTEIRO
        <FILENO>::= INTEIRO
    <lista-de-estruturas>::= (<nome externo>,,,)
    <marca-Pascal>::= <símbolo1>
    <comando>::=<classe><tam-codigo><tam-globais>
                        <id rotina><parâmetros>
        <classe>::=<início-FAZER>|<fim-FAZER>|<operativo>
        <tam-código>::= INTEIRO
        <tam-globais>::= INTEIRO
        <id rotina>::=<areano><fileno>
        <parâmetros>::=<nro-filhos><nro-parâmetros>
                      (<início-param><classe-param><parâmetro>...)
            <nro-filhos>::= INTEIRO  "número de entradas do
                                      comando"
            <nro-parâmetros>::= INTEIRO
            <início-param>::= <símbolo2>
            <classe-param>::= <c-arquivo>|<c-LDA>|<c-exp>|
                              <c-comando>|<c-TUPLIGCTL>|
                              <c-ENDER>|<c-LISTAPASCAL>|
                              <c-LISTACAMPOS>
            <parâmetro>::=<arquivo>|<LDA>|<expressão>|
                          <comando>|<TUPLIGCTL>|<ENDER>|
                          <LISTAPASCAL>|<LISTACAMPOS>
            <arquivo>::=<areano><fileno><tipoarq><tiporeg>
                        <registro>
              <tipoarq>::=<serial>|<relacional>|
                          <inversão>|<associativo>|
                          <temporário-cria>|
                          <temporário-del>|<PC>
              <tiporeg>::=<tupla>|<ligação>
              <registro>::=<reg-tupla>|<reg-ligação>
```

```
<reg-tupla>::=<nro-campos><campos>
    <nro-campos>::= INTEIRO
    <campos>::=(<tipo-campo><tam-campo>...)
       <tipo-campo>::=<inteiro>|<real>|
                      <char>|
                      <conjunto de char>|
                      <conjunto de inteiros>|
                      <conjunto de reais>|
                      <booleano>|<idreg>
          <tam-campo>::= INTEIRO
<reg-ligação>::=<ligante><ligado>
    <ligante>::=<nro-campos><campos>
    <ligado> ::=<nro-campos><campos>
<LDA>::=<tipo-LDA><corpo-LDA>
    <tipo-LDA>::=<LDA-tupla>|<LDA-ligação>
    <corpo-LDA>::=<LDA-corpo-tupla>|
                  <LDA-corpo-ligação>
       <LDA-corpo-tupla>::=<nro-campos>
                           <campos-detalhados>
       <LDA-corpo-ligação>::=<LDA-ligante>
                             <LDA-ligado>
          <LDA-ligante>::=<nro-campos>
                           <campos-detalhados>
          <LDA-ligado> ::=<nro-campos>
                           <campos-detalhados>
             <campos-detalhados>::=
                     (<tipo-campo><tam-campo>
                      <referência>...)
                <referência>::=<arquivo-origem>
                           <reg-no-arq><ordenação>
                           <campo-no-reg>
                           <sub-campo-no-reg>
                  <arquivo-origem>::=
                           <índice na TIR>
                  <reg-no-arq>::=<tupla>|
                                 <ligante>|
                                 <ligado>
```

```
                    <campo-no-reg>::=
                                    <posição-campo>
                    <ordenação>::=<ascendente>|
                                    <descendente>|
                                    <nenhuma>
<expressão>::= IMP <expressão><subexp1>|<subexp1>
  <subexp1>::= OU <subesp1><subexp2>|<subexp2>
      <subexp2>::= ET <subexp2><subexp3>|<subexp3>
        <subexp3>::= NÃO <subexp3>|<subexp4>
          <subexp4>::= (<expressão>)|<op.rel><elemento1>
                        <elemento2>|<função ELEM>
            <elemento1>::= <classe E1><E1>
            <elemento2>::= <classe E2><E2>
              <classe E1>::= <c-referência>|<c-lit-alfa>|
                              <c-lit-real>|<c-lit-int>
              <E1>::= <referência>|<literal>
                <literal>::= IDREG % do arquivo de li
                                    terais %
              <classe E2>::= <c-referência>|<c-lit-alfa>|
                              <c-lit-real>|<c-lit-int>|
                              <c-agregação>
              <E2>::= <referência>|<literal>|
                      <função agregação>
                <função agregação>::=<início-função>
                                    <agregação>
                                    <fim-função>
                  <agregação>::=<cont>|<card>|<soma>|
                                <media>|<desv>|<max>|
                                <min>
            <op.rel>::=<igual>|<diferente>|<maior>|
                      <menor>|<maior ou igual>|
                      <menor ou igual>
            <função ELEM>::=<início-função><elem>
                            <fim-função>
```

<TUPLIGCTL>::=<tipo-reg>

<ENDER>::= IDREG

<LISTAPASCAL>::=<nro-variáveis>(<nome-externo>,,,)

    <nro-variáveis>::= INTEIRO

<LISTACAMPOS>::=<nro-campos>(<posição-campo>...)

    <posição-campo>::= INTEIRO

ANEXO 4 IDENTIFICADORES PASCAL CRIADOS PELO GERADOR DE CÓDIGO

O módulo gerador de código declara, no programa do usuário, uma série de estruturas e rotinas. Os identificadores destes objetos são formados a partir de mnemônicos padrões seguidos de sufixos numéricos de 3 dígitos.

a) MBR <sufixo> : identificação de buffers e áreas temporárias;

b) MBF <sufixo> : identificação de descritores de arquivos;

c) MBO <sufixo> : identifica variáveis inteiras, parâmetros das primitivas MARK e RELEASE;

d) MBVL <sufixo> : identifica um registro global com variáveis locais de rotinas embutidas;

e) MBRG <sufixo> : registro de todas as variáveis globais geradas a partir de uma instrução LOBAN;

f) MBPROC <sufixo> : identificador de rotina que implementa comando não embutido;

g) MBOPEN <sufixo> : identifica rotina de inicialização de comando embutido;

h) MBGNEXT <sufixo> : identifica rotina de produção de resultados de comando embutido;

i) MBCLOSE <sufixo> : identifica rotina de encerramento de comando embutido;

j) MBRP <sufixo> : identifica ponteiro para um MBR <sufixo>;

k) MBFP <sufixo> : ponteiro para um MBF <sufixo>;

l) MBOP <sufixo> : ponteiro para um MBO <sufixo>;

m) MBVLP<sufixo> : ponteiro para um MBVL <sufixo>;

n) MBRGP <sufixo> : ponteiro para um MBRG <sufixo>;

o) MBESTADO : informa o estado final da última instrução LOBAN;

p) MBPROCF <sufixo> : identificador de rotina "filha" que será chamada pela rotina principal criada durante o processamento de um mesmo esqueleto.

ANEXO 5 EXEMPLO DE ROTINA PRÉ-PRONTA PARA IMPLEMENTAÇÃO

A seguir são apresentadas as rotinas pré-prontas, associadas ao operador LOBAN ESTREITAR, e as rotinas Pascal que podem resultar de seu preenchimento.

A primeira rotina é preenchida quando o resultado da operação deve ser gravado em um arquivo intermediário. A segunda é processada quando o operador aparece no CI4 como um parâmetro de outro comando (comando embutido) e deve produzir o resultado na base de uma tupla por vez.

Qualquer das duas rotinas pode ser preenchida de modo a ler dados de um arquivo (rotina não particionada) ou recebê-lo de outro comando (rotina particionada).

```
ESTREIT3I(*ARQ-ENT,ARQ-SAI,LDA-SAI):
                    %ESTREIT,ALGORITMO3,INDEPENDENTE%
                    %*ARQ-ENT pode ser um comando   %
                    %embutido se Filhos[1] > - 1    %
                    %Gera uma única procedure       %

PROCEDURE ESTREIT3I& INSERE-SUFIXO-PROC&;
  BEGIN
    WITH MBRGP& SUFIXO-PONTEIRO-GLOBAL&↑
                    %busca o contador de pointers%
    DO BEGIN
       MARK(MBOP & INSERE-MARK-NO(MARK)& );
                    %cria novo MARK-NO; seta novo MARK%
       &SEL-TIPOPARAM(COMANDO) (*NEW(MBFP & INSERE-SUFIXO
       (POSIÇÃO1 NA TIR)&↑); NEW(MBRP & INSERE-SUFIXO
       (POSIÇÃO1 NA TIR)&↑);)& %Se filho-E é NÃO PART%
       NEW(MBFP & INSERE-SUFIXO(POSIÇÃO2 NA TIR)&↑);
       NEW(MBRP & INSERE-SUFIXO(POSIÇÃO3)&↑); %Cria LDA-SAI%
       & INICIALIZA-FILEMAPS& %Inicializa filemaps se houverem%
       & SEL-TIPOPARAM(COMANDO) (MBOPEN & INSERE-SUFIXO-FILHO(11)&*OPEN
       (MBPF & INSERE-SUFIXO(POSIÇÃO1)&↑))&;
                        %Chama 'OPEN' ou OPEN(ARQ-ENT)%
```

```
OPEN(MBFP & INSERE-SUFIXO(POSIÇÃO2)&↑);
& SEL-TIPOPARAM(Comando) (MBGNEXT & INSERE-SUFIXO-FILHO(11) &
 (MBRP & INSERE-SUFIXO-FILHO(12)&, MBESTADO);
      *& SEL-TIPOARQ(WITH MBFP & INSERE-SUFIXO(POSIÇÃO1)&
         ↑.THISTUPLE
   DO BEGIN PAGENO:=Ø; CHARNO:=Ø END;
   SEQDATA*READ**)&(MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑,
                    MBRP & INSERE-SUFIXO(POSIÇÃO1)&↑);)&
WHILE & SEL-TIPOPARAM(comando) (MBESTADO  *MBFP & INSERE-SUFIXO
                                          (POSIÇÃO1)↑.RESULT
    <> EOFILE)&
DO BEGIN
   & MOVE3(3)& %move dados para LDA-SAI%
   WRITE(MBFP & INSERE-SUFIXO(POSIÇÃO2)&↑,
         MBRP & INSERE-SUFIXO(POSIÇÃO3)&↑);
   & SEL-TIPOPARAM(comando) (MBGNEXT & INSERE-SUFIXO-FILHO(11) &
                            (MBRP & INSERE-SUFIXO-FILHO(12)& ,
   MBESTADO);  *& SEL-TIPOARQ(SEQDATA*READ**)
               &(MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑,
                 MBRP & INSERE-SUFIXO(POSIÇÃO1)&↑);)&
         END;
   MBESTADO := FALSE;
   & SEL-TIPOPARAM(comando) (MBCLOSE & INSERE-SUFIXO-FILHO(11)&*
        CLOSE(MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑));
                  % fecha ARQ-ENT ou CLOSE-PROC %
   CLOSE(MBFP & INSERE-SUFIXO(POSIÇÃO2)&↑); % fecha ARQ-SAI %
   RELEASE(MBOP & INSERE-MARK-NO(RELEASE)&);
   END;
END;  % fim da procedure independente(completa) do ESTREIT3 %
```

```
ESTREIT3E(*ARQ-ENT,LDA-SAI):    % ESTREIT,ALGORITMO3,EMBUTIDO %
                                % *ARQ-ENT pode ser um comando %
                                % embutido se FILHOS[1] > - 1 %
                                & Gera 3 procedures: OPEN,GNEXT,CLOSE %

& GUARDA-INFO-FILHO &
PROCEDURE MBOPEN & INSERE-SUFIXO-PROC&;
  BEGIN
    WITH MBRGP & SUFIXO-PONTEIRO-GLOBAL &↑ DO
    BEGIN
      MARK(MBOP & INSERE-MARK-NO(MARK)&); % cria novo MARK-NO;
                                            seta novo MARK %
      &SEL-TIPOPARAM(comando) (*NEW(MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑);
                               NEW(MBRP & INSERE-SUFIXO(POSIÇÃO1)&↑);)&
                                          % Se ARQ-ENT é arquivo%
      NEW(MBRP & INSERE-SUFIXO(POSIÇÃO2)&↑);     % Cria LDA-SAI %
      & INICIALIZA-FILEMAPS& % inicializa filemaps se houverem %
      & SEL-TIPOPARAM(COMANDO) (MBOPEN & INSERE-SUFIXO-FILHO(11)&*
                               OPEN(MBFP & INSERE-SUFIXO(PSIÇÃO1)&↑);
                               & SEL-TIPOARQ(WITH MBFP & INSERE-SUFIXO
                                             (POSIÇÃO1)&↑.THISTUPLE
      % ABRE ARQ-ENT %                     DO BEGIN PAGENO:=Ø;CHARNO:=Ø END;
      % LÊ A 1ª TUPLA %                     SEQDATA*READ**)&
                              (MBFP & INSERE-SUFIXO(POSIÇÃO1)& ,
                               MBRP & INSERE-SUFIXO(POSIÇÃO1)&↑);
                               MBESTADO:=TRUE)&;
    END;
  END; % Fim do open %
```

```
PROCEDURE MBGNEXT & INSERE-SUFIXO-PROC&(VAR MBRP & INSERE-SUFIXO(POSI
                    ÇÃO2)&:UNIV POINTER2;VAR MBESTADO:BOOLEAN );
  BEGIN
    WITH MBRGP & SUFIXO-PONTEIRO-GLOBAL&↑ DO
    BEGIN
      & SEL-TIPOPARAM(comando)(MBGNEXT & INSERE-SUFIXO-FILHO(11)&
                               (MBRP & INSERE-SUFIXO-FILHO(12)&↑,
                               MBESTADO);*
      IF MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑.RESULT <> EOFILE
      THEN BEGIN
           & MOVE3(3)&; %Carrega LDA-SAI%
                           & SEL-TIPOARQ(SEQDATA*READ**)&
                             MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑,
                             MBRP & INSERE-SUFIXO(POSIÇÃO1)&↑);
           END
      ELSE MBESTADO:=FALSE;)&
    END;
  END; % Fim do GNEXT %

PROCEDURE MBCLOSE & INSERE-SUFIXO-PROC&;
  BEGIN
    WITH MBRGP & SUFIXO-PONTEIRO-GLOBAL&↑ DO
    BEGIN
    & SEL-TIPOPARAM(comando)(MBCLOSE & INSERE-SUFIXO-FILHO(11)&*
                           CLOSE(MBFP & INSERE-SUFIXO(POSIÇÃO1)&↑));
    RELEASE(MBOP & INSERE-MARK-NO(RELEASE)&);
    END;
  END; % Fim do CLOSE %
```

1) Comando embutido e não particionado:
   a) D=1; b) G=ØØ2; c) H1=2; d) Ixy não é usada;

   parâmetros: ARQ-ENT, LDA-SAI; CONT-PROCS=2;
                (01)   (02)

```
PROCEDURE MBOPENØØ2;
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
      MARK(MPOPØØ2);
      NEW(MBFPØØ1↑); NEW(MBRPØØ1↑);
      NEW(MBRPØØ2↑);
      WITH MBFPØØ1↑
      DO BEGIN
          AREANO:=A;
          FILENO:=F
         END;
      OPEN(MBFPØØ1↑);
      READ(MBFPØØ1↑, MBRPØØ1↑);
      MBESTADO:=TRUE
      END;
END;

PROCEDURE MBGNEXT(VAR MBRPØØ2:UNIV POINTER2; VAR MBESTADO:BOOLEAN);
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
      IF MBFPØØ1↑.RESULT<>EOFILE
      THEM BEGIN
           "MOVE MBRPØØ1↑ PARA MBRPØØ2↑";
           READ(MBFPØØ1↑, MBRPØØ1↑);
           END
      ELSE MBESTADO:=FALSE;
      END;
 END;
```

```
PROCEDURE MBCLOSEØØ2;
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
       CLOSE(MBFPØØ1↑);
       RELEASE(MBOPØ2);
       END;
  END;
```

2) Comando não embutido e não particionado:
   a)D=2; b) ↑=ØØ2; c) H1=2; d) Ixy não usada;
   parâmetros: ARQ-ENT; ARQ-SAI; LDA-SAI; CONT-PROCS=2;
                (Ø1)   (Ø2)   (Ø3)

```
PROCEDURE ESTREIT3IØØ2;
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
       MARK(MBOPØ2);
       NEW(MBFPØØ1↑); NEW(MBRPØØ1↑);
       NEW(MBFPØØ2↑);
       NEW(MBRPØØ3↑);
       MBFPØØ1↑.AREANO:=A1;MBFPØØ1↑.FILENO:=F1;MBFPØØ2↑.AREANO:=A2;
                                               MBFPØØ2↑.FILENO:=F2;
       OPEN(MBFPØØ1↑);
       OPEN(MBFPØØ2↑);
       READ(MBFPØØ1↑, MBRPØØ1↑);
       WHILE MBFPØØ1↑.RESULT<>EOFILE
       DO BEGIN
            "MOVE TO MBRPØØ3";
            WRITE(MBFPØØ2↑, MBRPØØ3↑);
            READ(MBFPØØ1↑, MBRPØØ1↑);
          END;
       CLOSE(MBFPØØ1↑);
       CLOSE(MBFPØØ2↑);
       RELEASE(MBOPØ2);
       END;
  END;
```

3) Comando não embutido e particionado:

a) D=2; b) G=ØØ2; c) H1=1; d) I1 1 = ØØ1; 2 = ØØ2 (outro ESTREIT);

parâmetros: ARQ-ENT; ARQ-SAI; LDA-SAI; CONT-PROCS=2;
(Ø1) (Ø2) (Ø3)

```
PROCEDURE ESTREIT3IØØ2;
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
       MARK(MBOPØ2);
       NEW(MBFPØØ2↑);
       NEW(MBRPØØ3↑);
       WITH MBFPØØ2↑ DO BEGIN AREANO:=A; FILENO:=F  END;
       MBOPENØØ1;
       OPEN(MBFPØØ2↑);
       MBGNEXTØØ1(MBRPØØ3↑, MBESTADO);
       WHILE MBESTADO
       DO BEGIN
            "MOVE TO MBRPØØ3";
            WRITE(MBFPØØ2↑, MBRPØØ3↑);
            MBGNEXTØØ1(MBRPØØ3↑, MBESTADO);
            END;
       MBCLOSEØØ1;
       CLOSE(MBFPØØ2↑);
       RELEASE(MBOPØ2);
  END;
END;
```

4) Comando embutido e particionado:

a) D=1; b) G=ØØ2; c) H1=1; I1 1=ØØ1 2=ØØ2(outro Estreit)

parâmetro: LDA-SAI;
(Ø2)

```
PROCEDURE MBOPENØØ2;
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
      MARK(MBOPØ2);
      NEW(MBRPØØ2↑);
      MBOPENØØ1;
      END;
END;

PROCEDURE MBGNEXTØØ2(VAR MBRPØØ2:UNIV POINTER2;
                     VAR MBESTADO:BOOLEAN);
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
      MBGNEXTØØ1(MBRPØØ2↑, MBESTADO);
      END;
END;

PROCEDURE MBCLOSEØØ2;
BEGIN
  WITH MBRGPØØn↑
  DO BEGIN
      MBCLOSEØØ1;
      RELEASE(MBOPØ2);
      END;
END;
```

BIBLIOGRAFIA

[ALL 76] ALLMAN, E. & STONEBRAKER, M. Embedding a relational data sublanguage in a general purpose programming language. IN: CONFERENCE ON DATA ABSTRACTION, DEFINITION AND STRUCTURE, Salt Lake City, Mar. 22-24, 1976. Proceedings. p. 25-35. (Em SIGPLAN NOTICES. v.11, special issue, 1976).

[BEL 73] BELL, J.R. Threaded code. Communications of the ACM, New York, 16(6):370-2, June 1973.

[BRI 77] BRINCH HANSEN, P. The arquitecture of concurrent programs. Englewood Cliffs, Prentice-Hall, 1977.

[BOR 82] BORAL, H.; De WITT, D.J. et alii. Implementation of the database machine DIRECT. IEEE Transactions on Software Engineering. New York, SE-8(8): 533-43. Nov. 1982.

[CAR 81] CARVALHO, M.A. Sistema operacional DUO. Porto Alegre, PGCC da UFRGS, 1981. (Não publicado).

[CAR 82] CARVALHO, M.A. & GOLENDZINER, L.G. Um sistema operacional para suporte a banco de dados. In: CONGRESSO NACIONAL DE INFORMÁTICA, 15., Rio de Janeiro, Out. 1982. Anais. Rio de Janeiro, SUCESU, 1982. p.137-45.

[CHA 76] CHAMBERLIN, D.D. et alii. SEQUEL2: a unified approach to data definition, manipulation and control. IBM J. Res. and Develop. 20(6):560-75, Nov. 1976.

[CHA 81] CHAMBERLIN, D.D. et alii. A history and evaluation of System R. Communications of the ACM, New York, 24(10):632-46. Oct. 1981.

[COD 70] CODD, E.F. A relational model of data for large shared data banks. Communications of the ACM, New York, 13(6):377-97, Jun. 1970.

[DAT 77] DATE, C.J. An introduction to database systems. 2.ed. Reading, Addison-Wesley, 1977.

[DUR 76] DURCHHOLZ, R. & RICHTER, G. Information management concepts (IMC) for use with DBMS interface. In: IFIP WORKING CONFERENCE ON MODELLING IN DATA BASE MANAGEMENT SYSTEMS, Freudenstadt, 1976. Proceedings. Amsterdan, North-Holland, 1976. p.49-69.

[GOL 80] GOLENDZINER, L.G. & HEUSER, C.A. Transformações para otimizar instruções em banco de dados. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE NACIONAIS, 7., Campinas, jul. 21-25, 1980. Anais. Campinas, UNICAMP, 1980. p. 45-52.

[GOL 82] GOLENDZINER, L.G. & HEUSER, C.A. Banco de dados: uma linguagem unificada. In: CONFERÊNCIA LATINOAMERICANA DE INFORMÁTICA, 9., Lima-Peru, Ago. 16-20, 1982. Anales. Lima, APCI, 1982. p. 527-37.

[GOL 83] GOLENDZINER, L.G.; HEUSER, C.A. & IOCHPE, C. Embutimento de LOBAN em PASCAL. Porto Alegre, CPGCC-UFRGS, 1983. (Projeto MINIBAN/UFRGS - Relatório técnico,31).

[HAW 79] HAWTHORN, P. & STONEBRAKER, M. Performance analysis of a relational data base management system. In: ACM-SIGMOD INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA, Boston, May 30 - June 1, 1979. Proceedings. ACM, New York, 1979. p.1-12.

[HEU 81] HEUSER, C.A.; GOLENDZINER, L.G. & OLIVEIRA,J.P.M. Sistema L: uma implementação da linguagem LOBAN. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 8., Florianópolis, Jul. 27-31, 1981. Anais. Florianópolis, UFSC, 1981. p.169-86.

[HEU 83] HEUSER, C.A. & GOLENDZINER, L.G. DABOL: a new data base language and its protable implementation. In: WORKSHOP ON RELATION DBMS DESIGN, IMPLEMENTATION, USE ON MICROCOMPUTERS, Toulouse, Feb.14-15, 1983. Proceedings. Rocquencourt, INRIA, 1983.

[IBM 82] IBM. SQL/Data System Concepts and Facilities: program product. 2.ed. Endicott, Feb. 1982 (GH24-5013-1).

[KAT 79] KATZ, R.H. Performance enhancement for relational systems through query compilation. In: NATIONAL COMPUTER CONFERENCE, ., New York, 1979. Proceedings. New York, NCC, 1979.p.741-7.

[LIM 82a] LIMA, J.V. & HEUSER, C.A. Um analisador semântico para a linguagem LOBAN. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, 9., Ouro Preto, Jul. 12-16, 1982. Anais. Rio de Janeiro, SBC, 1982. 15-27.

[LIM 82b] LIMA, V.L.S. & GOLENDZINER, L.G. Resolução de operações de consulta a banco de dados. In: CONFERENCIA INTERNACIONAL EN CIENCIA DE LA COMPUTACION, 2., Santiago-Chile, ago. 9-11, 1982. Actas. Santiago, Universidad de Chile, 1982. p.189-206.

[LOR 79] LORIE, R.A. & WADE, B.W. The compilation of a high level data language. San José, IBM, 1979. (IBM Research Report RJ 2598)

[MED 81] MEDEIROS, G.C.R. Implementação do sistema Pascal Concorrente no LABO8034. Porto Alegre, PGCC da UFRGS, 1981. (Dissertação de mestrado).

[PRY 83] PRYWES, N. & PNUELI, A. Compilation of nonprocedural specifications into computer programs. IEEE Transactions on Software Engineering. New York, SE-9(3):267-79, May. 1983.

[RIC 77] RICHTER, G. O sentido e o valor do banco de dados. Dados e Idéias, Rio de Janeiro,2(6):2-14, Jun./Jul. 1977.

[RIC 78] RICHTER, G., PEREIRA FILHO, J.C. & CASTILHO,J.M.V. Projeto MINIBAN: Relatório final da segunda etapa. Rio de Janeiro, DIGIBRÁS, 1978.

[RIC 81] RICHTER, G. Utilization of data access and manipulation in conceptual schema definitions. Information Systems, 6(1):53-62, 1981.

[RIS 77] RISNES, O. et alii. ASTRA - a DBMS based on a high level, relational DML with data access via a hierarquical DDL. In: SIMULA USERS CONFERENCE, Sept. 1977. Proceedings. apud KATZ, R.H. Performance enhancement for relational systems through query compilation. In: NATIONAL COMPUTER CONFERENCE, New York,1979. Proceedings. New York, NCC, 1979. p.742

[SAN 80] SANTOS, A.C. et alii. Especificação da interface LOBAN. Rio de Janeiro, COPPE/UFRJ,1980. (Projeto MINIBAN/COPPE - Relatório técnico).

[STO 76] STONEBRAKER, M. et alii. The design and implementation of INGRES. ACM Transactions on Database Systems, New York, 1(3):189-222, Sept. 1976.

[STO 77] STONEBRAKER, M. & ROWE, L.A. Observations on data manipulation languages and their embedding in general purpose programming languages. In: CONFERENCE ON VARY LARGE DATA BASE, 3., Tokyo-Japan, oct. 6-8, 1977. Proceedings. New York, IEEE, 1977. p.128-43.

[STO 80] STONEBRAKER, M. Retrospection on a database system. ACM Transactions on Database Systems. New York, 5(2):225-40. June 1980.

[STO 83] STONEBRAKER, M. et alii. Performance enhancements to a relational database system. ACM Transactions on Database Systems, New York, 8(2):167-185, June 1983.

[SMI 75] SMITH, J.M. & CHANG, P.Y.T. Optimizing the performance of a relational algebra database interface. Communications of the ACM, New York, 18(10):568-79. Oct. 1975.

[SUM 75] SUMMERS, R.C. et alii. A programming language extension for access to a shared data base. In: ACM PACIFIC CONFERENCE, Apr.1975. Proceedings. apud KATZ, R.H. Performance enhancement for relational systems through query compilation. In: NATIONAL COMPUTER CONFERENCE, New York, 1979. Proceedings. New York, NCC, 1979. p.742

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

UM ESQUEMA COMPILATIVO PARA LINGUAGENS
DE OPERAÇÕES DE BANCO DE DADOS

Dissertação apresentada aos Srs.

Antônio Luz Furtado

Clesio Saraiva dos Santos

José Mauro Volkmer de Castilho

Lia Goldstein Golendziner

Visto e permitida a impressão
Porto Alegre, 20 / 03 / 84

Coordenador do Curso de Pós-Graduação
em Ciência da Computação