UNIVERSIDADE FEDERAL DE MINAS GERAIS
COMISSÃO DE COORDENAÇÃO DAS ATIVIDADES
DE PROCESSAMENTO ELETRÔNICO-CAPRE

IV SEMINÁRIO SOBRE DESENVOLVIMENTO INTEGRADO DE SOFTWARE E HARDWARE

1977 VOLUME II

UNIVERSIDADE FEDERAL DE MINAS GERAIS
COMISSÃO DE COORDENAÇÃO DAS ATIVIDADES DE PROCESSAMENTO ELETRÔNICO - CAPRE/PNTC

IV SEMINÁRIO SOBRE DESENVOLVIMENTO INTEGRADO DE SOFTWARE E HARDWARE

VOLUME II

ORGANIZAÇÃO:

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO - ICEX-UFMG

CENTRO DE COMPUTAÇÃO DA UFMG

FUNDAÇÃO DO DESENVOLVIMENTO DA PESQUISA (FUNDEP)

COMISSÃO ORGANIZADORA:

PROF. JOSÉ MARCOS SILVA NOGUEIRA

PROF. NEWTON ALBERTO DE CASTILHO LAGES

COMISSÃO TÉCNICA

PROF. DALTRO JOSÉ NUNES

PROF. IVAN DA COSTA MARQUES

PROF. WILSON DE PÁDUA PAULA FILHO

LOCAL:

AUDITÓRIO DA REITORIA DA UFMG

DATA:

25 A 29 DE JULHO DE 1977

O CONCEITO DE DADO

APRESENTADOR: Atendolfo Pereda Borquez

AUTORES: Arndt Von Staa
Atendolfo Pereda Borquez

ENTIDADE: PUC/RJ - DI

Sumário

CAPÍTULO 1

INTRODUÇÃO

Nosso principal propósito neste trabalho é definir, em forma simples e precisa, uma serie de conceitos relacionados com o conceito de dado e com manipulação de dados, no contexto de ciencias de computação. Pretendemos com isto clarificar a estes conceitos. Vemos esta esquematização como um primeiro passo para uma formalização mais estrita e como uma tentativa de padronização de termos relacionados a esses conceitos.

Para caracterizar Dado, utilizaremos os dois aspectos fundamentais dele, que são aspectos Físicos e Aspectos Lógicos. Uma vez estabelecidas as relações entre os dois domínios: domínio físico e domínio lógico, estabeleceremos o que se entende por tipo de dado. Nos últimos dois capítulos investigaremos as formas de referência a dados e como passar de uma referência, ou seja, de a aparição de um nome simbólico num certo programa (conceito a nível sintático) ao acesso do dado desejado.

CAPÍTULO 2

O Conceito de Dado

Seções:

1- Aspectos físicos
2- Aspectos lógicos
3- Relação entre domínio físico e lógico

Computadores armazenam dados sob a forma de sequência de valores binários. Chamaremos bit à unidade de valor binário. Cada bit pode, portanto assumir os valores 0 ou 1 e somente este valores.

Dado é um conjunto de bits possuindo certas características físicas e lógicas. Como veremos mais tarde, um dado poderá ser armazenado em zero ou mais espaços de dado.

2.1 - Aspectos Físicos de Dados, Espaço de Dado.

Um dado precisa ser armazenável, o que se fará utilizando-se espaços de dados contendo parte ou todo o dado. Um espaço de dados localiza-se em um meio de armazenagem, tal como memória principal, disco ou fita magnética, registradores, cartões e mesmo listagens.

O espaço de dados deve ser acessível. Isto é, deve-se ter a capacidade de chegar até ele. Essa capacidade consegue-se através do conhecimento de sua locação (endereço).

São exemplos de locação: endereço da memória principal; < unidade de disco, "drive" de disco, cilindro, trilha, vetor> no caso de espaço de dado em disco; < unidade de fita, drive da fita , posição da fita sob o cabeçote de leitura> no caso de fitas magnéticas não blocadas ; Índice do registrador de propósito geral.

O espaço de dados ocupa um certo número de bits consecutivos, ou seja, um espaço no meio de armazenamento.

A tripla < meio,locação,espaço > define completamente um espaço de dados.

A título de exemplo, damos a seguir, descrições dos domínios físicos de diferentes estruturas de dados, que aparecem mais freqüentemente na prática:

a) Implementação tabular de matrizes: A[1:m, 2:n]

meio: Memória principal

locação: Endereço do primeiro elemento da matriz, por exemplo A[1,1] ocupa o endereço E.

espaço: Quantidade de memória ocupada pela matriz ou seja, se m é o número de linhas, e n o número de colunas, então, espaço é igual a m*n* tamanho do elemento

500	501	502	503	504	505	506
A[1,1]	A[1,2]	A[1,3]	A[2,1]	A[2,2]	A[2,3]	A[3,1]

507	508	509	510	511
A[3,2]	A[3,3]	A[4,1]	A[4,2]	A[4,3]

Fig. 1 Implementação tabular da matriz, A[1:4, 1:3] a partir do endereço 500

b) Implementação segmentada de matrizes

Nesta implementação, usa-se um "descritor principal", que é um vetor de descritores de segmentos, cada descritor de segmento contém um ponteiro para cada segmento em que é dividida a matriz, isto é cada descritor de segmento contém a Locação do segmento referenciado.

Cada segmento contém um número determinado de elementos da matriz. Logo na especificação do domínio físico, devemos considerar o domínio físico do descritor principal e dos segmentos.

i) Descritor Principal

meio : Memória Principal

locação: Endereço do primeiro elemento do descritor

espaço : É igual ao tamanho do descritor de segmento * número de segmentos.

ii) Para cada segmento

meio : Memória principal, ou disco, etc.

locação: Endereço do primeiro elemento do segmento dentro do meio.

espaço : Número de elementos * tamanho do elemento.

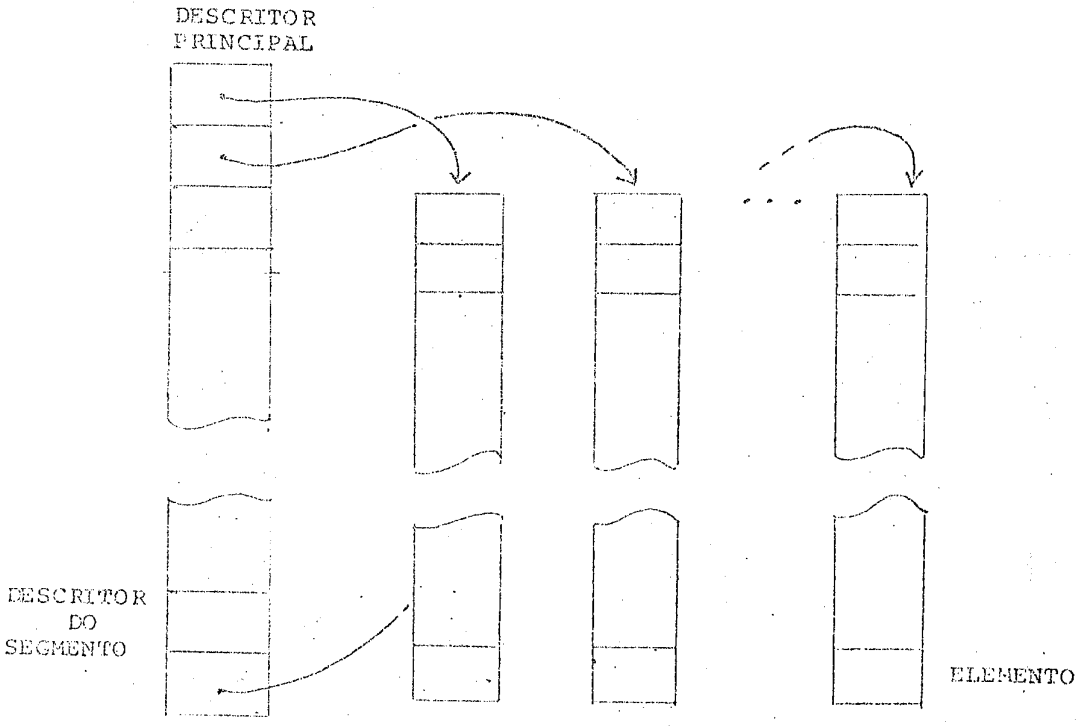

Fig.2.Implementação segmentada de matrizes

Além de conter a locação do segmento referenciado, o descritor do segmento poderia conter também a extensão do segmento referenciado. Ou seja o descritor visa a dupla < locação,extensão > que descreve um segmento (linha ou coluna) da matriz.

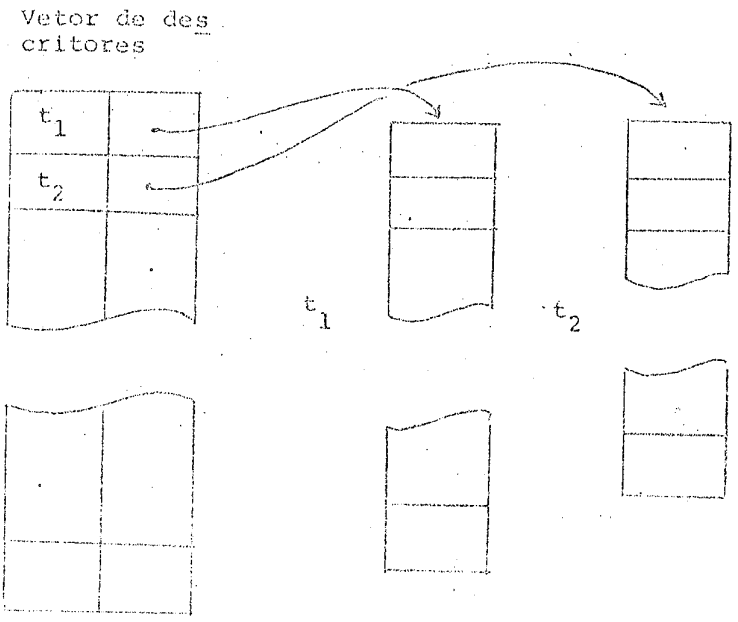

Fig. 3. Implementação segmentada de matrizes com número de elementos variáveis por segmento.

Na figura 3 mostramos pictòricamente como esta implementação de matrizes pode ser conseguida. A grande vantagem de uma representação desta natureza é a facilidade com que se pode controlar o acesso impedindo erros por extravasão.

Esta representação ainda não é completamente geral. Se o descritor de segmentos incorporar ainda o meio de armazenagem teremos a generalização completa. A título da informação, esta representação generalizada é a utilizada em diversos equipamentos para permitir a criação de memória virtual.

c) Implementação de Listas, usando áreas múltiplas, e vários nós por área

Neste caso, devem ser descritos três domínios físicos, correspondentes ao Descritor Principal, (que agora é um vetor de descritores de áreas, cada descritor de área contém um ponteiro para cada área), às áreas, que são espaços reservados em memória

principal ou outro meio, e aos nós componentes das listas.

i) Descritor Principal

meio : Memória principal

locação: Endereço inicial do descritor

Espaço : Número de descritores de áreas * tamanho do descritor

ii) Por área

meio : Memória principal, ou disco, etc.

locação: Endereço da área no meio

Espaço : Tamanho da área

iii) Por nó

meio : da área correspondente

locação: endereço na área

Espaço : tamanho do nó

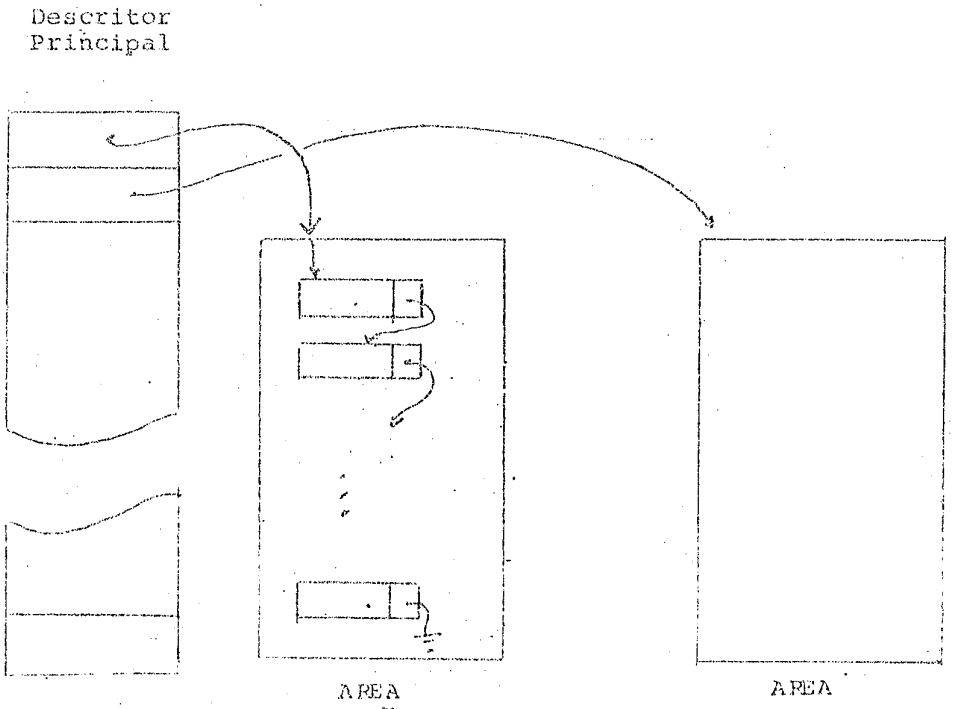

FIG. 4. Implementação de Listas

Esta forma de armazenar dados é utilizada em muitos sistemas da alocação dinâmica da memória.

2.2 - Aspectos Lógicos dos Dados

Cada dado é referido pelo usuário por um nome, que é uma cadeia de símbolos pertencentes a algum alfabeto, sujeita, esta cadeia, a regras sintáticas. O nome então, identifica o dado ou uma coleção de dados.

O nome pode ser simples ou composto, ou seja, qualificado, indicando características de estrutura do dado e/ou características estruturais da linguagem de programação em que é usado, e/ou ainda características dinâmicas do processo. A seguir damos alguns exemplos de nomes.

Exemplos de nomes simples

'X	identifica um elemento único
A[2,3]	identifica um elemento dentro dos elementos da matriz. A. A[2,3] é uma instância de A[inteiro, inteiro].
TAB<'NAME'>	identifica um dos elementos contidos em uma tabela SNOBOL. Esse exemplo é uma instância de TAB<string>

Exemplos de nomes qualificados

A qualificação pode ser explicita como em

A.B.C para identificar um elemento em uma estrutura em PL/I ou COBOL.

ou implícita, como em .

< instância atual do bloco para o qual X é local > . X em uma linguagem de programação com estrutura de blocos e permissão de instâncias múltiplas desses blocos (recursão, reentrancia).

Alguns casos de qualificação pouco comuns são os seguintes:

.Dinâmica (depende do tempo):
<tempo de criação do bloco>. LABEL ' [1]

A necessidade deste tipo de qualificação surge ao resolver-se problemas na obtenção do nome único (em tempo de execução) para variáveis LABEL. Na implementação de uma variável LABEL, geralmente o valor dela é dado por um par composto do endereço onde ocorre o label e o nível do bloco em que o label aparece; existem porém certos tipos de erros que não permitem a obtenção de nomes

únicos para labels com esse tipo de implementação, por exemplo, no programa a transferência conseguida por intermédio do GO TO referência em bloco não mais existente.

```
begin  label  LB;
    ---
  begin
   L: LB:=L;
  end;
    ---
  begin;
      begin;
          go to LB;
      end;
  end;
end;
```

Uma forma de resolver este problema é usar como um dos componentes do nome único o tempo de relógio da alocação do bloco no qual aparece o label. [1]

.Dependente de uma ação:

<read> .A

<write>.A

o nome único obtido só será válido se A está "autorizado" a ler (ou escrever).

.Dependente de um contador:

<número de acessos> . TRACEDVAR

Por exemplo, em SNOBOL-4[2], onde se proporcionam facilidades de seguimento da execução do programa usando o modo "trace". Quando esse modo está em efeito, podem testar-se mudanças do tipo, número de acessos a um certo label, etc.

Dentre todos os possíveis nomes que se podem formar (segundo as regras sintáticas) para designar dados, só um subconjunto deles é constituído de nomes que são referências válidas a dados. Ou seja, nem sempre existe um dado para um nome qualquer. Esse subconjunto de todos os nomes que constituem referências válidas a dados é o Domínio Lógico:

Exemplos de domínios lógicos, são os seguintes:

.D_1 = {'A[i,j]' | i,j ∈ N, 1≤i≤M, 1≤j≤N}

D_1 é o domínio de um array A[1:M,1:N] no ALGOL 60.

.D_2 = {'A(i,j)' | i,j ∈ N, 0≤(i*M*j-(M+1))*c≤rspace}

D_2 é o domínio de um array A(1:?,1:M) em FORTRAN, onde c é o tamanho do elemento, e rspace é o espaço sobrante na área de alocação do array. [4]

.D_3 = {'< data de expiração > ACCOUNT' | são ACCOUNT's tais que data atual≤ data de expiração}

.D_4 = {'< write >.A' | são A tais que o meio [A] não é protegido contra escrita}

Nos exemplos D_3 e D_4, devemos observar que o uso da qualificação não necessariamente implica em uma subindicação ao espaço físico dos dados, i.e., locação.

A sequência de bits que constitue um dado pode ser submetida a diferentes interpretações. Uma interpretação (nos níveis mais elementares equivale a decodificação) é um mapeamento dos bits, ou grupos de bits, que constituem o dado, a símbolos com significado próprio para o usuário e/ou computador. Por exemplo a sequência

1001 1000 1111 1001 1110 0010 1111 0000

no IBM/360 pode ser interpretada como o número hexadecimal

9 8 F 9 E 2 F 0

ou como os caracteres EBCDIC

q 9 s 0

ou como a instrução LOAD MULTIPLE

LM 15,9,752(14)

ou como o número em ponto flutuante, de precissão simples

$-(16^{-40} * 0.976\ 119\ 041\ 442\ 871\ 093\ 750)$

ou como o decimal inteiro em complemento a dois

- 1 728 453 904

A interpretação a que é submetido cada dado chamaremos de tipo de dado. Logo, o tipo de dado especifica o conjunto de valores válidos desse tipo, e as operações a que pode ser submetido.

A informação que se obtem de um dado, vem então, via o tipo desse dado. Evidentemente, poder-se-á ter interpretações (tipos) de diferentes níveis, por exemplo, um dado com interpretação (de nível baixo) "número inteiro", pode transportar mais informações se é interpretado a um nível mais alto como "números de peças produzidas". A introdução de novos níveis de interpretação, introduz redundância na linguagem que usa esses tipos, mas incrementa a segurança e a facilidade de deteção de erros semânticos [3].

Assim sendo, todos os dados usados num determinado momento deveriam ter, pelo menos, um tipo. Pode acontecer que um mesmo dado esteja sendo submetido a diferentes interpretações, i.e. que dois tipos diferentes sejam associados com um dado único. Por exemplo, por meio de um EQUIVALENCE em FORTRAN IV.

Um dado pode ser subdividido em vários "sub-dados". Ou seja, a tripla D = < meio,locação,espaço > pode subdividir-se em triplas d_i = < $meio_i$,$locação_i$,$espaço_i$ > para i=1,2,...,n, de tal forma que d_i esteja integralmente contido em D para todos os valores de i, i=1,2,...,n. Esta subdivisão chama-se "lay-out" do dado.

Por exemplo, em um vetor real A[1:20] teremos 20 "subdados" A[I] diferentes. Em uma estrutura PL/I, o lay-out descreve a disposição dos elementos da estrutura no espaço ocupado por toda a estrutura. Dessa forma, torna-se possível acessar os elementos a partir da estrutura completa.

Devido à existência dos tipos, os dados só aceitam subdados e não interseções. A existência do conceito de interseção de dados em algumas linguagens de alto nível , nada mais é que interpretações a nível mais alto de áreas multitipeadas.

O usuário ao referir-se a um dado, o faz utilizando o nome que identifica este dado. Necessita-se portanto de uma função que converta este nome à tripla < meio,locação,espaço > que caracteriza o dado. A esta função chamaremos de função de locação.

Existem vários níveis de função de locação. Por exemplo em uma máquina de memória absoluta quando programamos em linguagem absoluta (código de máquina) cada endereço absoluto E é simplesmente acrescido de memória principal e tamanho da palavra para gerar a tripla < memória principal,E,tamanho da palavra > e desta forma caracterizar o dado contido na palavra de endereço E da memória principal. Já quando houver indexação com o conteudo do registrador RX, temos como resultado da função de locação a tripla < memória principal,E+RX,tamanho da palavra >.

Quando programamos em linguagem simbólica (Assembler , PL/I, COBOL, etc.), a função ficará razoavelmente complicada e seu resultado será computado parcialmente durante o tempo de compilação, parcialmente durante o tempo de ligação e parcialmente durante o tempo de execução.

A diferença entre Layout e função de locação, reside em que o layout produz a informação para acessar um "sub-dado" de um dado (lista de parametros), enquanto que a função de locação computa a tripla < meio,locação,espaço > a partir dos valores fornecidos pelo layout.

Exemplo: A função de locação de uma estrutura de dados (PL/I ou COBOL) pode dar acesso a toda a estrutura ou então a porções dela. Para acessar porções da estrutura são necessários os valores da posição relativa contidos no layout.

Cada referência aos dados, usa uma certa quantia de recursos, gerando assim certo custo. O custo depende dos seguintes fatores:

a - Espaço ocupado pelo dado por unidade de tempo e por meio.

b - Dificuldade para acessar o dado, ou seja, o custo de avaliação da função de locação.

c - Espaço gasto pelo programador para definir e implementar a estrutura de dados.

Resumindo, temos como aspectos lógicos dos dados, os seguintes:

- O Nome
- O Domínio Lógico
- O Tipo
- O Layout
- A função de locação
- O custo

2.3 - Relação entre domínio físico e domínio lógico

Devemos observar que a palavra domínio tem sido usada com dois significados diferentes : na descrição do aspecto físico de dados:

domínio físico: < meio,locação,espaço >

e na descrição do domínio lógico.

Em geral, a satisfação do domínio lógico implica na satisfação do domínio físico. Existem porém, casos em que isto não

é válido. Por exemplo:

> Paginação: a satisfação do domínio lógico não implica em acesso imediato ao domínio físico.
>
> Tabela de símbolos: a satisfação do domínio lógico não implica que o elemento é referenciável, ou seja , que ele é fisicamente existente.

A interpretação dos diferentes aspectos lógicos de dados, é usada para testar consistência e validade desses dados a diferentes níveis. Por exemplo, pode dar-se o caso que o nome é uma referência válida, o dado existe, ou seja, está definido, mas o tipo do dado não é consistente com o tipo implicitado pelo nome que está fazendo a referência.

Outra inconsistência, detetada em forma mais simples, é no caso que o nome não é referência válida, por exemplo a referência A(21) em FORTRAN, se o array está declarado como A(20).

Uma terceira combinação possível, é que o nome seja válido, o tipo seja consistente, mas o dado (valor) é indefinido.

CAPÍTULO 3

TIPO DE DATO

Cada tipo de dado tem um conjunto de valores (válidos), i.e., configurações de bits, característico. A esse conjunto de valores chamamos conjunto-tipo do tipo de dado.

Ex. Se o tipo de dado é o LOGICAL de FORTRAN IV, então o conjunto tipo é:

{.TRUE., .FALSE.}

Se um tipo de dado é um tipo SCALAR de PASCAL definido como

(paus, ouros, copas, espada)

então o conjunto-tipo será:

{paus, ouros, copas, espada}

Se o tipo de dado é [ ]INTEGER em ALGOL 68, então o conjunto-tipo será: o conjunto de todos os valores possíveis com elementos inteiros.

Se se tem dois conjuntos tipos, pode-se fazer corresponder aos elementos de um os elementos do outro mediante certa função que chamaremos conversão.

Diz-se que um conjunto tipo é convertivel a um outro conjunto tipo se existe no primeiro um elemento que, sob, a função de conversão, é igual a um elemento do outro conjunto tipo.

Agora bem, o fato de que um conjunto-tipo seja convertivel a um outro, não quer dizer, que todos os elementos do primeiro conjunto tipo tenham um elemento correspondente no outro conjunto tipo sob a mesma função de conversão. Se esse for o caso, ou seja, se todos os elementos do primeiro conjunto tipo tem um elemento correspondente no outro conjunto-tipo, então se diz que o primeiro é sempre convertivel ao segundo conjunto-tipo.

No caso em que todo elemento do segundo conjunto-tipo é igual a algum elemento do primeiro, convertido pela função de conversão, então se diz que a função de conversão converte completamente o primeiro conjunto-tipo no segundo.

Se no processo de conversão se manter a configuração em bits de todo elemento convertivel do primeiro conjunto-tipo ao segundo, então se diz que existe uma conversão idêntica.

Ex. Seja o primeiro tipo valores absolutos de BIN FIXED (15,0) e o segundo tipo BIT (16). A função de conversão de BIN FIXED a BIT STRING, proporcionada por PL/I, será, nesse caso, uma função de conversão identica, sempre que a máquina permita inteiros de 16 bits.

O processo pelo qual, a certa configuração interna de bits dentro da máquina, i.e., a um certo dado, atribui-se um tipo,

recebe o nome de tipeamento, ou seja, tipeamento é a operação de associar um tipo a um dado. O tipeamento pode ser feito estatica - mente, i.e., antes da execução do programa, ou dinamicamente, i.e., durante a execução do programa.

O tipeamento dinâmica de dados não deve se confundir com a conduta dinâmica dos elementos de dados por exemplo, variáveis lo cais a um bloco de programa podem ser alocadas em diferentes partes da memória, cada vez que uma nova instância do bloco é criado, já sea devido as chamadas recursivas, ou a inicialização repetida da primeira ocorrencia do bloco, más, o tipo de cada variável não é mu dado em geral, sendo então assim tipeada estaticamente.

Ao falar de tipeamento dinâmico, implicitamente deve-se entender que existem períodos durante os quais um dado tem um tipo determinado e outros períodos nos quais tem um tipo diferente. O tempo durante o qual um dado tem um tipo fixo, é chamado intervalo de tipeamento, e esse intervalo começa imediatamente após do instan te em que se faz o tipeamento, até o instante imediatamente prece - dente a um novo tipeamento.

Da forma em que se efetua o tipeamento dinâmico, pode- se dizer que existem em geral duas, aquela na qual podem se ter vá- rias interpretações associados com um dado durante todo o processo mas, para cada unidade discreta de tempo na máquina, só tem um ti- po associado a ele. Esse é chamado dado SEQUENCIALMENTE MULTITIPE- ADO. Um caso particular dele é aquele do dado que durante todo o processo mantem o mesmo tipo, ou seja, não ocorrem retipeamentos . Essa forma é chamada TIPEAMENTO ÚNICO.

A outra forma permite que se tenham várias interpretações associadas com o dado, todas elas simultaneas durante um intervalo do tipeamento. Essa segunda forma recebe o nome de PARALELAMENTE MULTITIPEADO.

Nas linguagens de alto nível, os dados são normalmente ti peados, já seja explicitamente, como em ALGOL 68, PL/I, FORTRAN, AL GOL 60, etc., ou implicitamente como em SNOBOL, GEDANKEN. As lin - guagens nas que ocorre tipeamento implicito, como as duas últimas a costumam sex chamadas linguagens sem tipos, (type-less languages), más, essa terminologia não é a mais adequada como se pode observar do ditos dito. Nelas, os tipos são asignados pela linguagem a ca- da dado, e não explicitamente pelo usuário.

CAPÍTULO 4

FORMA DE REFERÊNCIA

Para que o dado seja de utilidade para o usuário, este deve dispor de mecanismos que lhe permitam ter acesso ao dado. Quando falamos de acesso de um dado, queremos significar qualquer operação na qual se faz referência ao dado ou a uma parte dele. Logo, acesso ao dado serão as operações de "fetch" e "store", como também as operações de E/S. O termo Acesso é mais usado no contexto do nível de máquina. No contexto de linguagens de programação, se usa o termo REFERÊNCIA para significar o mesmo que acesso. Logo, uma referência, em geral, é associada com um nome simbólico que se referirá ao dado acessado.

Dependendo dos testes feitos com tipos dos dados sendo acessados por o uso de uma referência, podemos dizer que se tem duas formas de referências:

a) Referência com (teste de) conversão:

Nela, a referência aos dados produz um teste de que tipo do dado acessado esteja correto, e, em caso necessário, se fazem as converções de tipos adequados. Esse teste do tipo de um dado pode ser realizado em tempo de compilação, ou seja, previo à execução do programa, ou pode-se realizar durante o tempo de execução.

b) Referência sem (teste de) conversão:

A referência é feita usando um tipo determinado mais não se realizam testes para detetar se ocorreu erros por diferenças entre o tipo de dado acessado e o tipo exigido na referencia. Por exemplo, essa forma de referência é comum na maioria das linguagens montadoras ("assembler"). O acesso a um dado multitipeado em paralelo, não significa, implicitamente, que a forma de referência (ao nome desse dado) seja referência sem (teste de) conversão. O multitipeamento em paralelo permite vários tipos, mas se realiza teste de tipos nas referências.

Ex.: Considere-se os seguintes comandos FORTRAN, que aparecem em duas subrotinas diferentes (de um mesmo sistema):

```
COMMON / A / IX     na subrotina SUB1
```

e

```
COMMON / A / AX     na subrotina SUB2
```

No capítulo anterior definimos tipos de dado como a interpretação a que é submetido cada dado. Essa interpretação fixa o conjunto de valores válidos desse tipo, as operações à que estes valores podem ser submetidos, e a forma de criar valores desse tipo a partir de valores de outros tipos.

Olhando para essa definição, é imediata a observação que existirão níveis de tipo, dependendo da menor ou maior complexidade da representação do conjunto de valores válidos e/ou as operações que com eles podem efetuar-se.

Sendo assim, sempre existirá um nível mais elementar de tipos de dados, que dependerá exclusivamente da máquina sendo usada no sentido que serão tipos para os quais se tem instruções de máquina válidos para eles. A esse tipo de dado chamaremos de tipo primitivo de dado. Evidentemente, o conjunto de tipos primitivos dependerá da máquina.

Nem sempre cadeias de bits são tipos primitivos. Por exemplo, as cadeias de bits genéricas não são um tipo primitivo no IBM/360, mas o são no HONEYWELL 6000 quando equipado com Extended Instruction Set. Uma caracterização intuitiva é a de que um tipo primitivo necessita de somente uma instrução para ser processado enquanto que se não for primitivo necessita de mais de uma.

A partir dos tipos de dados primitivos, podem-se obter tipos de dados composto, fazendo a associação lógica de zero ou mais tipos de dados, cada um dos quais pode ser, por sua vez, um tipo de dado primitivo ou um tipo de dado composto. (É lógico que em algum momento se terá um tipo de dados composto, formado só de um ou mais tipos de dados primitivos). Um exemplo muito simples, é uma estrutura PL/I.

```
DCL    1A,
         2B,
           3C  BIN  FIXED,
           3D  BIN  FIXED,
         2E  BIT(15);
```

Segundo nossa definição, seria possível a existência de tipos de dados compostos, nos quais, um (ou mais) dos subtipos compostos constituintes se contivessem a se mesmos como componentes. Se nenhum dos subtipos constituintes do tipo de dado tem a característica de se conter a se mesmo como componente, se diz então que esse é um tipo de dado composto acíclico.

Em SUB1, IX é um inteiro (se não existe declaração explícita em contra), em SUB2,AX é considerado um real (a mesma observação anterior). Esse é um caso evidente de multitipeamento em paralelo, porque em qualquer instância de tempo o dado elementar apontado por IX e AX pode ser referenciado. Agora, se as normas ANSI[4] fossem aplicadas rigorosamente, esse exemplo de multitipeamento em paralelo reduzir-se-ia a um multitipeamento sequencial, porque, segundo elas, se o último "store" feito fosse de tipo inteiro, todos os "fetches" subsequentes até o próximo "store" (excluindo esse) tem que ser do tipo inteiro, sendo a situação identica para o caso do tipo real.

Agora, como exemplo de forma de referência sem (teste de) conversão, seja o seguinte conjunto de comandos em "assembler" do IBM 7044:

```
CLA  A        limpe o acumulador e some o conteudo de A a
              ele.
FAD  B        soma (flutuante) do conteudo de B ao acumula-
              do.
STO  C        armazene o conteudo do acumulador em C.
 :
A.  BCI  1,-4000    Corresponde a 41. em ponto flutuante.
B.  DEC  -17246978048    constante decimal (corresponde
                         -1. em ponto flutuante).
C   OCT  0     constante octal.
```

Sem preocuparmos com a semântica do código acima, podemos ver que eles são comandos perfeitamente aceitáveis pelo assembler. A interpretação de cada nome é dado exclusivamente pelo acesso e não pelas declarações desses nomes.

Segundo todo o dito anteriormente uma linguagem de programação que permite referencias com (teste de) conversão, deve conter um conjunto (ordenado) de tipos primitivos e/ou compostos, definidos pela linguagem, ou pelo usuário, segundo suas necessidades. Ex.: PASCAL, ALGOL 68, etc. [5][8] Para cada nome usado na linguagem (i.e. identificador que não seja palavra chave), deve existir um tipo fixo associado, se é de tipeamento simples, ou um conjunto válido de tipos de dados, cada um deles válido se é usado em multitipeamento sequencial, ou um conjunto valido de tipos de dados, válidos simultâneamente, se são usados em multitipeamento em paralelo. Nes

se último caso deve-se dispor de um critério de seleção de qualificação, para definir qual é o tipo dominante dentre eles. Evidentemente, uma linguagem contendo essas características, deve dispor para realizar os testes de conversão de tipos nas referências de:

- uma função de teste de tipos, função que determinará o tipo (atual) ou tipo dominante do nome (dado elementar).
- uma função de tipeamento, que fixa o tipo (atual) ou tipo dominante do nome (dado elementar), e
- de funções de conversão de tipos, associadas a um predicado que testa se a conversão entre dois tipos (diferentes) é possível usando uma dada função de conversão. Os resultados que podem surgir mediante o uso dessas funções de conversão e do predicado, são os seguintes.
  - a conversão é válida, não existem problemas na sua realização.
  - Erro-, os dois tipos dedados não são conversíveis através dessa função de conversão.
  - Erro - o tipo de dado a ser convertido a um segundo tipo de dado pela função de conversão não está no domínio desta. Ex.: '1A2' a inteiro.
  - Erro - o tipo de dado a ser convertido, é convertível em princípio, mas seu valor atual não é representável no tipo de dado que é o rango da função de conversão.
    Ex.: '1234' a inteiro de 8 bits.

CAPÍTULO 5

FUNÇÃO DE LOCAÇÃO E "DOPE DATA"

Para poder ter acesso aos dados, é preciso conhecer onde eles estão, ou seja, é preciso conhecer o endereço da locação na memória, onde o dado está armazenado. Para o usuário, basta dar o nome do dado para fazer uma referência a ele, o sistema tem que se encarregar de testar se o nome é válido para ter acesso ao dado,e, se é válido, calcular o endereço.

A função que computa a locação (o endereço) de um dado referenciado pelo nome N é chamada Função de Locação. O predicado cujo objeto é testar se o nome N é válido para ter accesso ao dado recebe o nome de Predicado de locação da função de locação específica de que se trate.

Para poder operar, a função de locação e o predicado de locação aceitam informação através de diversos parametros:

i) Parâmetros actuais-formais

Especificam um elemento particular de um dado composto

Ex.: os subscritos de um array

ii) Parâmetros Intrinsicos

São sub-elementos de sistemas compostos que mais tarde conterão o elemento referenciado.

Ex.: Um apontador ao próximo elemento numa lista.

iii) Parâmetros locais

São parte do espaço de dados da função de locação Se pode ter accesso a eles através de funções apropriadas.

Ex.: Os limites de array de tamanho fixo.

iv) Parâmetros globais

São parte do medio no qual a função de locação existe.

Ex.: o "display" de ALGOL 68.

Cada referência feita pelo usuário de um certo nome, é submetida a dois testes, associados com o domínio do dado referenciado:

i) Teste de parâmetros: Testa se o nome pertence ao domínio de nomes válidos

ii) Teste de Espaço: Testa se o locador e a área de armazenamento do dado referenciado estão totalmente dentro do segmento asignado de armazenamento.

Normalmente, se o teste de parâmetros aceita o nome, então o teste de espaço da também resultado correto. Existem casos onde essa implicação não é verdadeira; Ex.: no medio onde se realiza alocação dinâmica de memória, pode acontecer que não exista espaço suficiente para armazenar os novos dados.

Um dado pode ser referenciado por medio de várias funções de locação diferentes. Por outra parte, o dado pode ser composto e requerer uma descomposição adequada para poder ser manipulado pelo computador em que será armazenado e elaborado. Essa é particularmente importante quando se refere a transmissão de argumentos, porque a rotina que recebe os argumentos deve ter a informação que o dadó que recebe é composto.

No contexto, estamos usando o termo "parâmetro" num sentido amplo e mais geral que o acostumado, pois entendemos por parâmetros informação imediatamente disponível, ex.: números, ou uma porção de código que após execução, produz a informação desejada, ex.: Uma função.

Ao conjunto de parâmetros locais usados pela função de locação para calcular a locação de algum dado, damos o nome de "Dope Data".

Uma subclasse de "dope data" são os "dope vectors" [6]. Não temos adotado essa terminologia, porque não exigimos que a informação sobre os parâmetros seja fisicamente armazenada como um vector nem nos confinamos ao uso exclusivo de valores como é normal no caso dos "dope-vectors". Nos exemplos que daremos mais tarde, usaremos, por motivos de simplicidade, uma representação de tipo vector.

Dependendo da compacticidade da totalidade dos dados sendo accessado, se podem realizar duas formas de accesso.

i) Accesso em bloco: Nele, é accessado (referenciado) um dado composto ou primitivo (nome) completo.
   Ex.: Accesso de uma substrutura PL/I, ou uma fila em ALGOL 68.

ii) Accesso disperso: Nele são accessados (referenciados) uma coleção de 0 ou mais itens relacionados de um dado composto (nome)
   Ex.: referência à diagonal de um array de duas dimensões.

Na maioria dos casos, os dados são armazenados em áreas contiguas de memória, logo, accesso em bloco significe o accesso a

toda essa área, ou a uma parte contigua nela; accesso disperso é o accesso a sub-seções que, em geral, não são contiguas nessa área . Por exemplo, asumamos que uma lista (completa) é uma instância de accesso em bloco (pelo nome), apesar do fato qué a lista provavel - mente não ocupa áreas contíguas na memória. Nesse caso, accesso disperso é accessar algum conjunto de sublistas mediante algum artifício que não é simplesmente o nome duma sublista, i.e.

A: (BC: (DE) F: (GH))

Accesso em bloco é referencias A, ou B, ou C, etc., já que cada um desses nomes representa uma seção (logicamente) contigua da lista; accesso disperso será o uso da lista X:(A,C,H), ou seja, a referencia a uma coleção de sublistas que não estão (logicamente) contiguas.

É evidente que accesso em bloco é uma subclasse de accesso disperso, temos dado um nome especial a ele, porque é a forma mais comum de accessar dados, e existem linguagens onde é a única forma de accesso. Ex.: ALGOL-60 se o truque de programação conhecido como "Jensen's device" não é usado [7]

A fim de poder explicar melhor o rol dos "dope data" , daremos a seguir alguns exemplos, mas, para esse é necessário estabelecer primeiro uma terminologia Estandar na qual todos eles se - rão apresentados.

Seja a figura seguinte, a representação de um "layout" seqüêncial do "dope data".

	L	S		n		
Entrada 1	A_1	D_1	σ_1	Z_1	V_1	T_1
Entrada 2	A_2	D_2	σ_2	Z_2	V_2	T_2
⋮	⋮	⋮	⋮	⋮	⋮	⋮
Entrada n	A_n	D_n	σ_n	Z_n	V_n	T_n

As L, S, n, e A_i $i=1,n$, são definidas no momento de definição do dado. O significado dos diferentes símbolos é o seguinte:

L = locação da área de dados
S = espaço alocado para a área de dados
n = número de entradas no "dope data"
λ_i = índice da entrada antecessora da entrada i (o pai)

D_i = Predicado de domínio da entrada i

σ_i = função que computa o deslocamento da entrada i com respeito à entrada antecessora de i, se A_i não é nula, ó com respeito a L se A_i é nula.

Z_i = função que computa o espaço ocupado pela entrada i

V_i = função que computa o espaço interpretável pela entrada i.

T_i = função de teste de tipo, retorna o tipo da entrada i.

Toda a informação necessária para completar o "dope data" não está completa em tempo de compilação (ou em tempo de execução, se é possível a existência de declarações dinâmicas).

Ex.:1) "Dope Data" para o array

A[1:10, 2:2, 3:7] integer (usando notação ALGOL 68)

L	10*2*5		3		
→	$1 \le A_1 \le 10$	(A_1-1)	4*2	4*2	[,] integer
↕	$2 \le A \le 3$	(A_2-2)	4	4	[ ] integer
↕	$3 \le A_3 \le 7$	(A_3-3)	1	1	integer

"Dope data" de

Ex.:2) Um "slice" A[a:b, c:d, e:f] do array anterior

ref Ex1	ref. Ex.1		3		
→	$a \le A_1 \le b$	(A_1-1)	4*2	F_1	[,] integer
↕	$c \le A_2 \le d$	(A_2-2)	4	(c-d)	[ ] integer
↕	$e \le A_3 \le f$	(A_3-3)	1	1	integer

Ex.:3) "Dope data" da estrutura PL/I

```
DCL  1 A(10),
       2  B (J,K),
          3  C  CHAR(1),
          3  D  CHAR(2),
       2  E  CHAR(3);
```

L	10*((J*K)*3+3)		6		
→	$1 \le A_1 \le 10$	(A_1-1)	(J*K*3)+3	(J*K*3)+3	struct
↑	$1 \le A_2 \le J$	(A_2-1)	3*K	3*K	struct
↑	$1 \le A_3 \le K$	(A_3-1)	3	3	struct
↓	true	0	1	1	char 1
	true	1	1	2	char 2
	true	(J*K)*3	1	3	char 3

Referências

[1] Fenichel, R.R. "On Implementation of label variables" C.A.C.M vol. 14, Nº5, 1971

[2] Griswold, R.E. et alii "The SNOBOL 4 programming languages" Prentice-Hall, Insc. 1968

[3] Cleaveland, J.C. "Pouches, a programming language construct encauraging redundance". U.C.L.A. Report. 1975

[4] IBM Corporation FORTRAN IV language

[5] Van Winjgaarden, A. et alii. "Revised Report on the algorithmic language ALGOL 68", Acta Informática, vol 5, Fasc. 1-3, 1975

[6] Gries, D. "Compiler Construction for Digital Computers" John Wiley & Sons, Insc. 1971.

[7] Froberg, E. "ALGOL", Oxford University Press. 1971

[8] Wirth, N. "The Programming Language PASCAL" Acta Informática, vol.1, Fasc. 1, 1971

CURRICULUM VITAE

Atendolfo Pereda Borquez, Engenheiro Químico pela Universidade de Concepcion - Chile - 1968. Mestre em Informática pela PUC/RJ - 1975. Professor no Departamento de Informática - PUC/RJ.