PUC

Série: Monografias em Ciência da Computação
Nº 12/82

INTRODUÇÃO À PROGRAMAÇÃO COM TIPOS ABSTRATOS DE DADOS

por

Francisco E. P. Pessoa

e

Paulo A. S. Veloso

Departamento de Informática

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO DE JANEIRO
RUA MARQUÊS DE SÃO VICENTE, 225 – CEP-22453
RIO DE JANEIRO – BRASIL

Serie: Monografias em Ciência da Computação
Nº 12/82

Editor: Marco A. Casanova Dezembro 1982

INTRODUÇÃO À PROGRAMAÇÃO COM TIPOS ABSTRATOS DE DADOS*

por

Francisco E. P. Pessoa+
e
Paulo A. S. Veloso

* Pesquisa realizada com auxílio financeiro da FINEP e do CNPq

+ Em licença do Núcleo de Processamento de Dados, Universidade Federal do Ceará, Fortaleza, CE.

RESUMO

Este trabalho se propõe a dar uma introdução didática a alguns aspectos importantes de programação com tipos abstratos de dados, já que estes têm sido apontados como uma poderosa ferramenta no desenvolvimento de programas confiáveis. O emprego de tipos abstratos de dados proporciona uma elegante construção do programa por fatorá-lo em duas partes: um programa que manipula objetos do tipo abstrato de dados por meio de suas operações e uma implementação, que representa os objetos e operações deste tipo em termos de outros mais concretos.
Assim se obtém uma fatoração natural de várias tarefas de programação: especificação, desenvolvimentos, documentação, verificação, testes, etc.

No capítulo introdutório tecem-se considerações sobre o emprego de abstrações em programação, programação por refinamentos sucessivos, etc. Em seguida, são apresentados os mecanismos de estruturação de dados propostos por Hoare e como eles aparecem em algumas linguagens de programação, como PASCAL, ALGOL 68 , etc. Além disso, são tratados mecanismos de abstração de uso geral, como os existentes na linguagem CLU. É esta a linguagem utilizada no capítulo III para apresentar um exemplo detalhado ilustrando o processo de desenvolvimento de programas utilizando tipos abstratos de dados. O capítulo final contém algumas considerações sobre especificações formais e implementações de tipos abstratos de dados.

PALAVRAS CHAVES: tipos abstratos de dados, mecanismos de estruturação de dados, desenvolvimento de programas, linguagens de programação.

ABSTRACT:

This paper presents a didactical introduction to some important aspects of abstract data types in programming. Abstract data types have been recognized as a powerful tool in the development of reliable programs. The usage of abstract data types enables an elegant program construction: the program is factored into two parts; a program manipulating abstract data types objects by means of their operations and an implementation, which represents these objects and operations in terms of more concrete data types. As a result a natural factorization is obtained for several programming tasks, namely specification, development, documentation, verification, testing, etc.

The introductory chapter consists of general remarks about the usage of abstraction in programming, stepwise refiments, etc. The next chapter presents Hoare's data structuring mechanisms and how they appear in programming languages like PASCAL, ALGOL 68, etc. It also presents general purpose data abstraction mechanisms such as those existing in CLU. This programming language is employed in chapter III to present a detailed example illustrating the process of program development by means of abstract data types. The final chapter contains some general remarks on formal specification and implementation of abstract data types.

KEY WORDS: abstract data types, data structuring mechanisms, program development, programming languages.

SUMÁRIO

CAPÍTULO I

INTRODUÇÃO

O conceito de tipo é de fundamental importância em linguagens de programação. Qualquer programador sabe que num programa, cada variável, constante ou expressão tem associado a ela um único tipo. Em PASCAL, por exemplo, a associação de um tipo a uma variável é feita pela sua declaração de tipo; em FORTRAN, tal associação é deduzida da primeira letra do nome da variável, caso esta não tenha sido explicitamente declarada.

Um tipo de dados é basicamente uma coleção de valores munida de um conjunto de operações e testes. O tipo Boolean, em PASCAL, por exemplo, tem como valores o conjunto {false, true} e como operadores {V, Λ, ¬} para indicar a disjunção "ou", conjunção "e" e negação, respectivamente. Embora tipos diferentes possam ter o mesmo símbolo de operador (exemplo: + para os tipos inteiro e real) numa linguagem, tais símbolos são interpretados como operações diferentes, podendo essa diferença ser detectada em tempo de compilação.

Associado a um tipo também está uma representação para os valores do tipo, uma vez que, sem uma representação, não teríamos como descrever o comportamento das operações referentes a ele. A representação do tipo inteiro numa linguagem de programação está implícita, de modo que, o programador não se preocupa

com ela , mas as operações a nível de máquina foram definidas em termos dessa representação. Assim sendo, a especificação da representação também determina o conjunto de valores de um tipo , uma vez que ela é definida em termos de tipos existentes (o tipo inteiro é representado por meio de um conjunto finito de bits, e seus operadores definidos com base nessa representação).

O conceito de tipo não é uma novidade surgida com a teoria das linguagens de programação. Na realidade os matemáticos e lógicos estão familiarizados com esse conceito. No raciocínio matemático é comum fazer-se distinção entre funções reais , funções complexas, conjunto de funções etc. De fato, quando um matemático introduz no seu raciocínio uma nova variável, ele costuma ter o cuidado de explicitar seu tipo.

Exemplos:

"Seja f uma função de duas variáveis reais"

"Seja S uma família de conjuntos de inteiros"

Abstração se refere ao processo mental pelo qual, quando em confronto com um conjunto de objetos, situações ou processos, podemos reconhecer suas similaridades, concentrando-nos sobre estas e ignorando suas diferenças. Para citar um exemplo , consideremos o conjunto {7, 1.33, 'A', 0.5 E6} que contém valores dos três tipos providos por quase toda linguagem de programação : inteiro, real e caráter. Olhando o conjunto mais abstratamente podemos dizer que ele contém apenas dois tipos: número e caracter . De um ponto de vista mais abstrato ainda, o conjunto tem apenas um tipo: constante.

Comum a todas as ciências, em ciência da computação, contudo, o termo abstração tem adquirido, gradualmente, um significado mais específico, ao se referir ao desejo de se querer considerar um conceito independentemente de suas várias possíveis representações.

Um tipo abstrato de dados é suposto ser independente de sua representação, no sentido de que os detalhes de como ele é implementado são propositadamente "escondidos" de seus usuários. O usuário de um tipo abstrato é provido com certas operações para o tipo, necessitando saber apenas o que tais operações fazem e não como elas o fazem. O comportamento do usuário deve ser semelhante àquele que ele tem para com o uso de uma sub-rotina, que descreve uma função específica por meio de um algorítmo que lhe é desconhecido: no ponto em que ele invoca a sub-rotina, o detalhe relevante é o que a sub-rotina faz, sendo o como totalmente irrelevante. Da mesma forma, ao nível de implementação " é desnecessário complicar o como por considerações dos porquês, isto é, as razões para a invocação da sub-rotina não precisam ser consideradas por seu implementador" (Guttag [12]).

De um certo modo, os tipos providos por uma linguagem de programação (ex. inteiro, real, etc.) são abstratos, já que o programador faz uso deles sem se preocupar com sua representação. Porêm o termo tipo abstrato de dados tem sido reservado para aqueles tipos que não são supridos pela linguagem, mas criados pelo usuário, fazendo uso dos mecanismos que a linguagem lhe oferece para esse fim. Tais tipos, normalmente, são usados num nível e realizados noutro mais baixo. Mas isso não se dá au

tomaticamente. Ao invés disso, um tipo abstrato de dados é realizado ao se escrever um programa que o define em termos da suas operações.

As linguagens de programação convencionais oferecem uma poderosa ferramenta para construção de abstrações que são as funções ou procedimentos. Linguagens como ALGOL 68 e PASCAL oferecem mecanismos que possibilitam a definição e uso de novos tipos. Em muitos casos, o novo tipo é definido em termos de outros previamente definidos, chamados de tipos constituintes ; os valores do novo tipo são estruturas de dados que tem como componentes os valores dos tipos constituintes, os quais podem ser selecionados e extraídos da estrutura. SIMULA 67 (Dahl-Hoare [03]) introduziu o mecanismo de classe que permite a definição de tipos abstratos de dados em termos das operações aplicáveis aos valores desse tipo. Esse mecanismo foi aperfeiçoado nas linguagens CLU (Liskov-Zilles [22] , [23] , [25]) e outras, como ALPHARD (Wulf-London-Shaw [36] , [37] , [33] e mais recentemente ADA (Ledgard [20]).

As características salientes do conceito de tipo que mais interessam à comunidade de computação são assim sumarizadas (Hoare [17]):

i) Um tipo determina a classe de valores que pode ser assumido por uma variável ou expressão.

ii) Todo valor pertence a um único tipo.

iii) O tipo de um valor denotado por uma constante, variável ou expressão pode ser deduzido de sua forma ou contexto, sem qualquer conhecimento prévio de

seu valor.

iv) Cada operador espera receber operandos de algum tipo fixo, e devolve um resultado de tipo fixo (usualmente o mesmo). Quando um mesmo símbolo é aplicado a diferentes tipos, esse símbolo pode ser visto como ambíguo, denotando diferentes operadores. A resolução de uma tal ambigüidade sempre pode ser conseguida em tempo de compilação.

v) As propriedades dos valores de um tipo e as operações primitivas definidas sobre eles são especificadas formalmente e de modo independente de representação por meio de axiomas.

vi) A informação de tipo numa linguagem de programação de alto nível é usada para prevenir ou detectar construções desprovidas de sentido num programa e para determinar o método de representação e manipulação de dados no computador.

Abstração é um processo inerente as aplicações dos computadores ao mundo real. A primeira atitude do programador quando do projeto de qualquer programa é concentrar-se nas características relevantes do problema, ignorando os fatores considerados irrelevantes. O uso de linguagens tradicionais exige que o passo seguinte do projeto seja a decisão de como representar no computador a informação abstrata. O passo final e então programar o computador para fazê-lo manipular as representações de dados de forma que essas manipulações levem ao efeito desejado no mundo real.

Confiabilidade e facilidade de entendimento são duas qualidades desejáveis de um programa . A programação estruturada é uma tentativa de disciplinar o processo de construção de programas de modo a se obter programas com essas características. De acordo com essa disciplina, um problema é resolvido por meio de um processo de sucessivas decomposições (Wirth [35]). Num primeiro passo o programador escreve um programa que resolve o problema, mas que opera sobre objetos abstratos, no sentido de que os mesmos não estão definidos na linguagem que será usada na codificação do programa (a conveniência disso está no fato de que o programador escolhe os objetos e operações que mais se adequem a solução do problema).

A tarefa seguinte do programador é procurar se convencer de que seu programa resolve corretamente o problema. Nesse ponto ele deve se preocupar apenas de como seu programa faz uso das abstrações, e não com qualquer detalhe de como elas serão realizadas. Quando satisfeito com a correção de seu programa o programador volta sua atenção para as abstrações feitas. Cada abstração representa um novo problema, requerendo solução. Essa solução pode ser dada em termos de novas abstrações. O problema estará completamente resolvido quando todas as abstrações forem realizadas na linguagem escolhida.

Como se depreende facilmente, o método de desenvolvimento de programas por refinamentos sucessivos, descrito acima, estimula o programador a deixar de lado a decisão de como representar os dados, para dedicar-se a elaboração de seu algorítmo, expresso como um programa abstrato, operando sobre dados abstratos. Isto feito, o programador escolhe para os dados uma repre

sentação concreta em termos de tipos diretamente ou quase diretamente representáveis na memória do computador, programando as operações primitivas requeridas pelo programa abstrato. Essa forma de fatorar a construção de programas num programa que manipula um tipo abstrato de dados e numa implementação do tipo abstrato em termos de uma representação selecionada, alivia, além de tornar construtiva, a tarefa de verificação de correção do programa, também fatorada na verificação de correção do programa abstrato e na verificação de correção da representação de dados.

Como se pode notar, a programação com tipos abstratos de dados segue naturalmente a idéia de desenvolvimento de programas por refinamentos sucessivos, subvertendo a atitude tradicional de construção de programas descrita (abstração → representação → codificação).

No que se segue enfocaremos alguns aspectos concernentes a programação com tipos abstratos de dados. O capítulo II é devotado ao estudo dos mecanismos atualmente existentes nas linguagens de programação que permitem a manipulação de tipos abstratos de dados. Basicamente são apresentados os mecanismos supridos pelas linguagens do tipo ALGOL-68, PASCAL e CLU. No capítulo III apresentaremos, por meio de um exemplo, uma metodologia para construção de programas que manipulam tipos abstratos de dados. Para isso faremos uso da notação introduzida no capítulo II. Finalmente, no capítulo IV são feitas algumas breves considerações sobre a especificação e a implementação de tipos abstratos de dados, tópicos que são tratados com mais detalhes em [32] .

CAPITULO II

MECANISMOS DE ABSTRAÇÃO EM LINGUAGENS DE PROGRAMAÇÃO

II.1 - Introdução

Uma linguagem de programação convencional, geralmente , provê o usuário com um conjunto de tipos chamados primitivos, e algum ou alguns mecanismos que aplicados a estes geram agrupamentos, as chamadas estruturas de dados. Para citar um exemplo, ALGOL 60 tem os tipos inteiro, real e lógico como primitivos, e como único mecanismo de estruturação de dados o vetor (array).

É indiscutível a influência que uma linguagem de programação exerce sobre a metodologia de desenvolvimento de programas. Numa linguagem convencional o programador procura determinar de antemão que abstrações da linguagem ele pode aplicar em seu programa. Evidentemente isto não é desejável pois que assim o programador está se preocupando com a representação, antes de ter uma idéia exata da solução do problema. O desejável, e que está de acordo com a metodologia de desenvolvimento de programas pelo método dos refinamentos sucessivos, é que o programador possa livremente escolher as abstrações que mais se adequem a solução do seu problema. Isso, naturalmente, requer da linguagem de programação, não apenas mecanismos para construção de algumas estruturas, mas também um mecanismo geral que permita estender a linguagem para conter as abstrações desejadas pelo usuário.

Desde o aparecimento de SIMULA 67 (Dahl-Hoare [03])que tem surgido um número cada vez maior de linguagens com mecanismos voltados para o uso de abstrações, no que concerne a estruturação e manipulação de dados. Dentre essas estão ALGOL 68 (Lindsey-Meulen [21]) e PASCAL (Jensen-Wirth [19]), que se enquadram entre aquelas linguagens que possuem os mecanismos de estruturação de dados limitados a certas estruturas, e CLU (Liskov-Snyder-Atkinson-Schaffer [25]) e ALPHARD (Wulf-London- Shaw [37]), que, além de possuirem mecanismos dessa natureza, aceitam de forma natural qualquer abstração desejada pelo usuário.

Nas seções que se seguem vamos apresentar, inicialmente, uma discussão dos mecanismos de estruturação de dados apresentados por Hoare [17] , ilustrando sua implementação em linguagens como SIMULA 67, ALGOL 68 e PASCAL. Em seguida apresentaremos um mecanismo mais geral de abstração de dados, implementado em linguagens como CLU e ALPHARD.

II.2 - Mecanismos de Estruturação de Dados

Hoare [17] formulou uma teoria de estruturação de dados que consiste na especificação de um certo número de métodos padronizados para definição de tipos de dados e no seu uso na declaração de variáveis de um programa.

Em geral, a definição de um novo tipo é feita a partir do "nada" ou em termos de outros tipos previamente definidos, chamados de tipos constituintes. A definição de um tipo a

partir do "nada" significa acrescer à linguagem um tipo cujos valores são unidades indivisíveis denotadas por literais. Tais tipos devem ser vistos como semelhantes àqueles normalmente supridos pela linguagem, os tipos primitivos (ex.: inteiro, real), e junto com estes constituem os chamados tipos escalares ou, como quer Hoare, não-estruturados. Por seu turno, uma estrutura de dados é um novo tipo definido em termos de outros tipos previamente definidos, sejam estes por sua vez estruturados ou não. Em última análise, contudo, os componentes de uma estrutura pertencem aos tipos não-estruturados.

Enquanto um tipo escalar é visto como uma unidade indivisível (embora sua representação possa ser por meio de uma estrutura; ex.: o tipo inteiro nada mais é que uma estrutura de bits), um tipo estruturado pode ter seus elementos constituintes selecionados e extraídos da estrutura.

II.2.1 - Tipos Não-Estruturados

Normalmente, uma linguagem de programação supre o usuário com um certo conjunto de tipos não-estruturados (ex.: inteiro, real). Embora esses tipos sejam "teoricamente adequados para todos os propósitos, existem fortes razões práticas para encorajar um programador a definir seus próprios tipos não-estruturados, quer para tornar clara sua intenção acerca do conjunto de valores que uma variável pode assumir, e dar uma interpretação a tais valores, quer para permitir um projeto subsequente de uma representação mais eficiente" (Hoare [17]).

Enumeração

É comum em programação o uso de associações como: 1-masculino; 2-feminino. Numa tal situação os inteiros 1 e 2 não são, ou não devem ser, vistos como uma quantidade, mas sim como seletores para um conjunto razoavelmente pequeno de alternativas, cuja interpretação é estabelecida pelo programa que os usa. É desejável, assim , ver-se tais valores como pertencentes a um tipo diferente de inteiro . Isso é possível através do mecanismo de construção de tipos, chamado enumeração.

Uma enumeração é um tipo cujos valores são caracterizados pelo conjunto finito de constantes daquele tipo. Das linguagens abordadas, apenas PASCAL possui mecanismo para sua construção, que tem a forma

$$\text{type } T = (K_1, K_2, \ldots\ldots, K_n)$$

onde T é o nome do novo tipo e K_1, K_2, ..., K_n são constantes, os nomes dos valores do tipo.

Para o exemplo citado acima poderíamos definir o seguinte tipo:

```
type sexo = (masculino, feminino)
```

Outros exemplos:

```
type cor = (vermelho, azul, branco)
type diasemana = (segunda, terça, quarta,
                  quinta, sexta, sabado, domin
                  go)
```

A declaração de uma variável é denotada por

var v : T

onde v é o identificador da nova variável, enquanto T é seu tipo.

Na presença das declarações

var feriado: diasemana

var corquente: cor

os seguintes comandos de atribuição são perfeitamente válidos

feriado:= domingo

corquente:= vermelho,

sendo inválidos feriado := vermelho e corquente := domingo

O tipo enumeração como definido tem os elementos de seu conjunto de valores não somente distintos como também ordenados. Valem então os seguintes axiomas:

a) $K_i \neq K_j$, para $i \neq j$ (unicidade)

b) $K_i < K_j$, para $i < j$ (ordenação)

c) somente $K_1, K_2, \ldots K_n$ são valores do tipo T

d) $Succ(K_i) = k_{i+1}$, para $i = 1,2,\ldots, n-1$

e) $Pred(K_i) = K_{i-1}$, para $i = 2,3,\ldots, n$

Pred (predecessor) e Succ (sucessor) são funções da linguagem que mapeiam um tipo escalar nele próprio, usando para tal sua ordem implícita (ex.: Succ(azul) tem valor branco).

PASCAL não dispõe de mecanismo destacado para definição de novas operações sobre os tipos definidos por enumeração, de modo que se o usuário as deseja, deve apelar para o procedimento.

Subdomínio

PASCAL dispõe de um outro mecanismo para definição de um tipo não estruturado, o Subdomínio. Um subdomínio consiste de um subconjunto conexo de algum tipo discreto previamente definido. Por exemplo:

```
type digito = 0..9
type fimdesemana = sabado .. domingo
```

define digito como o subconjunto { 0,1,2,... 8,9} dos inteiros e fimdesemana como o subconjunto { sabado, domingo } do tipo semana definido anteriormente.

Em PASCAL, a única linguagem dentre as mencionadas aqui que tem o mecanismo para construção de subdomínios, tal mecanismo pode ser aplicado a qualquer tipo escalar, exceto reais.

Hoare [17] discute os problemas associados com subdomínios e apresenta uma axiomática para os mesmos.

II.2.2 - Tipos Estruturados

Definir um novo tipo em termos de outros, primitivos ou previamente definidos, é uma tarefa algo semelhante àquela des

crita para enumeração. A diferença básica reside no fato de que o conjunto de valores do novo tipo não é explicitamente especificado, mas o modo como a estrutura é descrita, determina seus componentes como valores de tipos conhecidos. As regras de especificação da estrutura tem importância fundamental na facilidade com que o programador pode transformar suas abstrações em representações concretas.

Hoare [17] especificou os seguintes mecanismos de estruturação de dados: Vetor, Produto Cartesiano, União Discriminada, Conjunto (powerset), Sequência e Estruturas Recursivas.

Vetor

O vetor é a estrutura de dados mais familiar a muitos programadores e em algumas linguagens de programação é a única estrutura disponível (ex.: ALGOL 60).

Um vetor, de um ponto de vista abstrato, é um mapeamento entre um domínio de um tipo (o conjunto de valores dos índices) e um conjunto de valores de outro tipo qualquer (os elementos do vetor). Matematicamente:

$$A:D \rightarrow R$$

onde D é o tipo do domínio e R é o tipo do conjunto de valores (numa notação mais familiar aos programadores: tipo A = vetor D de R).

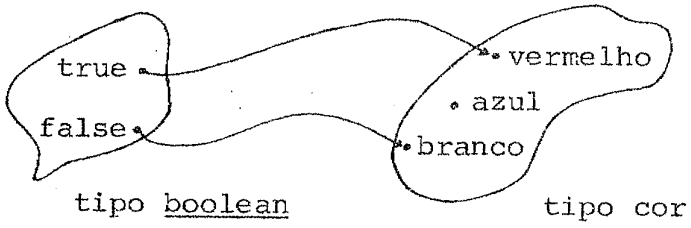
Em PASCAL, a definição do tipo vetor tem a forma

$$\underline{\text{type}}\ V = \underline{\text{array}}\ [T_1]\ \underline{\text{of}}\ T_2$$

onde V é o identificador do novo tipo, T_1 denota o tipo do índice do vetor e T_2 o tipo de suas componentes. Um exemplo de declaração de variável é

$$x : \underline{\text{array}}\ [\underline{\text{boolean}}]\ \underline{\text{of}}\ \text{cor}$$

que tem o seu significado ilustrado abaixo (boolean é o tipo primitivo lógico e cor é o tipo definido em II.2.1):



x[true] = vermelho

x[false] = branco

Uma vez que os valores de um vetor podem ser de qualquer tipo, é possível se especificar um vetor de vetores, que é um modo conveniente de se ver um vetor multidimensional. A declaração

$$y: \underline{\text{array}}\ [1..10]\ \underline{\text{of}}\ \underline{\text{array}}\ [1..5]\ \underline{\text{of}}\ \underline{\text{real}}$$

é um vetor bidimensional (matriz 10 x 5) de reais.

Infelizmente, em PASCAL os limites dos índices de um vetor devem ser constantes em tempo de compilação, o que impossibilita representações às vezes desejáveis, como por exemplo, um

vetor de vetores com tamanhos variáveis. Em ALGOL 68, contudo, isso é possível, conforme o seguinte exemplo

```
flex [1:0] real w
```

que declara w como sendo um vetor, inicialmente com zero elementos, cujo índice tem limite inferior 1 e o superior é variável. Esta construção permite que w tenha tamanho variável. Seu uso, porém, está restrito ao comando de atribuição, que tem no seu lado esquerdo um vetor de qualquer tamanho. Exemplo:

```
w := (5, 8, 6.35, -4.2)
```

Uma matriz triangular poderia ser declarada com esse mecanismo, da seguinte forma

```
[1: n] flex [1:0] real w
```

A maior desvantagem apresentada pelo ALGOL 68 é o fato de os índices do vetor terem de ser inteiros. Suas vantagens são os vetores poderem ter fronteiras flexíveis e poderem ser dinamicamente alocados.

SIMULA 67 é muito semelhante a ALGOL 68, com a diferença que as fronteiras do vetor não podem ser flexíveis.

Produto Cartesiano

Comumente utilizado em matemática, o produto cartesiano é um mecanismo de agrupamento de objetos de tipos diferentes

da mesma forma que o vetor agrupa objetos de um mesmo tipo. A definição do referido tipo obedece ao seguinte esquema:

$$\underline{\text{tipo}}\ T = (s_1:T_1;\ s_2:T_2;\ldots;s_n:T_n)$$

onde T é um identificador para o novo tipo, s_1, s_2, ..., s_n são os identificadores (variáveis) que atuam como seletores dos componentes do tipo T, e T_1, T_2, ... , T_n são os tipos dos componentes.

Exemplos:

```
tipo complexo = (partreal: real; partimag: real)
tipo pessoa = (nome:string; idade:inteiro; sexo:(mas
                culino, feminino))
```

O primeiro exemplo define complexo como sendo um par ordenado de reais, correspondentes a sua parte real e imaginária, respectivamente. No segundo exemplo string é um tipo previamente definido, enquanto sexo esta sendo definido por enumeração.

Em PASCAL, o tipo registro (record) é equivalente ao produto cartesiano. Nessa linguagem os exemplos anteriores seriam assim codificados:

```
type complexo = record partreal : real;
                       partimag : real
                end;
type pessoa = record nome: string ;
                     idade: integer ;
```

```
                    sexo: (masculino,feminino)
end
```

O tipo string não é primitivo na linguagem, havendo assim a necessidade de definí-lo antes de usá-lo na definição do tipo pessoa. Isso poderia ter sido feito com o mecanismo de vetor, assim:

```
type string = array [1..30] of char
```

Caso Tibério seja uma variável declarada como do tipo pessoa, seus constituintes (campos) seriam referenciados por Tibério.nome, Tibério.idade e Tibério.sexo. As seguintes atribuições seriam válidas:

```
Tibério.nome [9] := 'P'
Tibério.idade    : = 4
Tibério.sexo     : = masculino
```

Em ALGOL 68, o produto cartesiano é definido como uma estrutura (struct). Os exemplos apresentados seriam aqui codificados como,

```
mode complexo = struct (real partreal,partimag)

mode pessoa = struct (string nome, int idade,
                      bool sexo)
```

Observe que string é um tipo primitivo em ALGOL 68, e que, pelo fato de a linguagem não aceitar a definição de um tipo

por enumeração, foi feito um mapeamento entre um tipo conhecido (no caso o bool) e o conjunto { masculino, feminino } .

A referência aos campos da estrutura é feita de modo semelhante ao PASCAL, diferindo apenas na sua sintaxe. Assim, se Jorge é do mode pessoa, são válidas as seguintes atribuições:

```
nome of Jorge : = "Jorge Henrique Pinheiro"
idade of Jorge: = 2
sexo of Jorge : = true
```

O conceito de classe (class), o mais importante mecanismo de abstração em SIMULA 67, implementa o produto cartesiano, como indicado nos exemplos seguintes:

```
class complexo;
    begin real partreal, partimag end;

class pessoa;
    begin text nome; integer idade; boolean sexo end;
```

A seleção dos campos da estrutura é feito de modo semelhante a PASCAL.

União Discriminada

É comum se definir um conjunto de objetos como sendo a união de conjuntos previamente definidos. Embora o conceito de união se aplique a qualquer conjunto, estamos particularmente interssados na união de conjuntos disjuntos. Assim sendo,

dado um elemento qualquer da união deve existir alguma propriedade que nos permita identificar sua origem. Para citar um exemplo simples, considere o conjunto das pessoas, que pode ser visto com constituído da união de dois conjuntos: o conjunto dos homens e o conjunto das mulheres. Nesse caso a propriedade que descrimina uma pessoa é o sexo.

A definição de tipo para objetos pertencentes a uma união descriminada, obedece ao seguinte esquema:

$$\underline{\text{tipo}}\ T = (p_1{:}T_1,\ p_2{:}T_2, \ldots\ldots\ldots, p_n{:}T_n)$$

onde T é o identificador do novo tipo, $p_1, p_2, \ldots, p_n$ são os identificadores (seletores) das propriedades que permitem descriminar um elemento do tipo, e $T_1, T_2, \ldots, T_n$ são os tipos dos possíveis componentes da estrutura. O importante a notar é que um dado objeto do tipo T tem uma e apenas uma das propriedades $p_1, p_2, \ldots, p_n$.

No exemplo apresentado acima (Flon [08]), o conjunto das pessoas é visto como formado da união dos conjuntos de homens e mulheres. Se assim o fazemos é porque estamos interessados em observar propriedades diferentes para os elementos desses dois conjuntos. Vamos supor que essas propriedades sejam: Para os homens, sua altura e seu peso, e para as mulheres a cor dos olhos. Podemos assim definir os seguintes tipos, onde sexo é uma união descriminada:

```
tipo masculino = (peso:inteiro; altura:real)
tipo feminino  = (claro, escuro)
tipo sexo  =  (homem:masculino, mulher:feminino)
```

Poderíamos, agora, definir o tipo pessoa como o produto cartesiano (um registro) de nome, idade e sexo:

tipo pessoa = (nome:string; idade:inteiro; s:sexo)

Exemplos de objetos do tipo pessoa seriam:

(Malu, 24, claro)

(José, 26, (70,1.71))

Em PASCAL, a união discriminada é implementada como um registro. O exemplo citado seria assim codificado:

```
type pessoa = record nome: string;
                     idade: integer;
                     case sexo:(masculino, feminino)of
                     masculino:(peso:integer;altura:real);
                     feminino:(corolhos:(claro,escuro))
end
```

Observe que sexo é aqui um campo da estrutura, que tem este e mais nome e idade como comuns. O registro fica completo com uma das alternativas da cláusula case determinado pelo valor da variável sexo.

O tipo string não é da linguagem e precisa ser definido como o fizemos no item anterior.

Em ALGOL 68, a implementação da união descriminada é feita diretamente através do mecanismo union. O exemplo em anã

lise ficaria, nessa linguagem:

```
mode masculino = struct (int peso, real altura);
bool feminino;
mode sexo = union(masculino, feminino);
mode pessoa = struct(string nome, int idade, sexo
                     s);
```

Embora o sexo não faça parte da estrutura é possível, dado um objeto p do tipo pessoa, recuperar-se essa informação pela construção seguinte:

```
case s of p in
    (masculino): ..............;
    (feminino) : ..............
esac
```

Em SIMULA 67 a união discriminada pode ser implementada através da concatenação de classes.

Conjunto

A potência de um conjunto é definido como sendo um conjunto cujos elementos são todos os subconjuntos do conjunto dado. Como exemplo, seja o conjunto E = { f, e, p } . A potência de E, denotado por P(E), é o conjunto { ∅, {f},{e},{p},{f,e} , {f,p},{e,p}} (Se n é a cardinalidade de um conjunto E, P(E) tem cardinalidade 2^n).

A definição de um tipo conjunto estabelece uma coleção

de objetos que são os subconjuntos de um conjunto base. O esquema para definição desse tipo é o seguinte:

tipo T = conjunto de T_1

onde T é um identificador para o novo tipo e T_1 é o tipo base do qual se deseja subconjuntos.

PASCAL, ao contrário de SIMULA 67 e ALGOL 68, tem esse mecanismo de definição de tipo. Exemplo:

```
type corbasica = (vermelho, azul, amarela)
type cor = set of corbasica
```

Se laranja é uma variável do tipo cor, a seguinte atribuição é válida:

laranja : = [vermelho, azul]

PASCAL oferece ainda um conjunto de operadores (união, intersecção, pertinência, etc.) para objetos desse tipo.

Sequência

É conveniente dispor-se de um tipo cujos objetos tenham um número arbitrário de itens de um determinado tipo. Exemplos comuns são listas, "strings" de caracteres, pilhas, arquivos,etc. O mecanismo de sequência fornece esta facilidade.

Ao contrário das estruturas até então apresentadas, um tipo sequência tem cardinalidade infinita, uma vez que se admite um número arbitrário de itens. Esse fato faz com que a quantidade de memória alocada para uma variável desse tipo não seja determinada pela simples declaração desta. Hoare [17] discute os problemas de representação surgidos com tais tipos.

PASCAL tem um mecanismo semelhante ao de sequência, chamado de arquivo (file). A definição de um tipo file é assim feita:

type F = file of T

onde F é o identificador do novo tipo que é um arquivo de objetos do tipo T.

Exemplo:

type texto = file of char

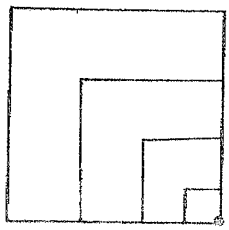
A declaração de f como variável do tipo file introduz automaticamente uma outra variável (buffer variable), denotado por f↑. Essa variável pode ser considerada uma "janela" através da qual podemos inspecionar (ler) ou acrescer (escrever) novos componentes ao arquivo.

Recursividade

A ocorrência de um tipo como parte de sua própria definição faz com que uma instância do tipo contenha como seu com

ponente uma instância "menor" do próprio tipo. Tipos definidos desse modo são ditos recursivos e têm um comportamento análogo àquele de um procedimento recursivo .

Estruturas recursivas são comuns e alguns exemplos são as definições de árvores binárias e expressões aritméticas. Para ilustrar no entanto o conceito, considere a figura abaixo:



Por essa figura cada quadrado representa um objeto de um determinado tipo, em que um de seus campos é por sua vez um objeto do mesmo tipo. O ponto no quadrado mais interno significa um objeto vazio. Formalmente:

tipo figura = (vazio, quadrado)

tipo quadrado = (s_1:T_1; s_2:T_2; ...; s_n:T_n; próximoquadrado: figura)

A utilização de apontadores (pointers) possibilita a definição de estruturas recursivas em SIMULA 67, ALGOL 68 e PASCAL.

II.3 - Mecanismos de Abstração de Dados de Uso Geral

Conforme já dissemos no capítulo I, um tipo de dados é uma coleção de objetos munida de um conjunto de operações e testes. A expressão tipo abstrato de dados refere-se ao fato de querermos ver os objetos do tipo independentemente de sua representação. Dito de outra forma, o comportamento do tipo é para ser caracterizado unicamente por suas operações.

A definição de um tipo por meio das operações que o caracterizam dá ao programador, quando do desenvolvimento de um programa, plena liberdade na escolha das abstrações que mais se adequem à solução do problema. Essa liberdade, desejável, não é permitida pelos mecanismos de estruturação de dados apresentados em II.2, por, dentre outras, as seguintes razões:

i) As operações para manipulação de uma estrutura se restringem a seleção dos seus componentes, aos quais, então, são aplicados as operações pertinentes.

ii) As operações para manipulação de uma estrutura são pré-definidas na linguagem (Ex.: em PASCAL, o tipo set dispõe, entre outras, das operações +, * e - para união, intersecção e diferença de conjuntos, respectivamente) e qualquer operação diferente destas deve ser definida por meio de um procedimento.

iii) O conjunto de operações fornecido pela linguagem para manipulação de uma estrutura pode ser amplia do através de um construtor de operações (Ex.: op em ALGOL 68), porém tais operações são genéricas no sentido de que elas não estão ligadas a defini ção do tipo.

Uma forma de se associar um tipo com suas operações foi razoavel - mente resolvido em SIMULA 67 com o mecanismo de class, uma vez que procedimen tos e dados podem ser declarados como parte de um class. Desse modo, o meca- nismo de class reune, num só lugar, todas as informações relevantes de uma par ticular implementação de um tipo abstrato (encapsulamento).

Liskov e Zilles ([23] , [25]) aprimoraram o concei to de classe de SIMULA 67 e propuseram a noção de cluster da lin guagem CLU, enquanto Wulf, London e Shaw ([33] , [36] , [37]) fizeram o mesmo com a linguagem ALPHARD, onde o conceito é conhe cido como form. Para ilustrar esse conceito, utilizaremos, basi- camente, a sintaxe de CLU.

II.3.1 - Abstrações em CLU

A linguagem de programação CLU foi projetada para acei tar o uso generalizado de abstrações. Isso significa dizer que a metodologia de desenvolvimento de programas por refinamentos su cessivos é naturalmente realizada na linguagem. Aliás, esse é um dos motivos apontados pelos seus autores para o seu desen volvimento.

CLU incorpora os três mecanismos de abstração apontados como importantes nos trabalhos sobre metodologia de programação: procedimentos, dados abstratos e estruturas de controle. Os procedimentos são realizados de forma semelhante as outras linguagens.

Dados Abstratos

Um tipo abstrato, caracterizado por um conjunto de operações, é, como já frisamos, implementado por meio do construtor linguístico cluster, que define um módulo externo independente.

Para ilustrar a definição de um cluster, considere que no curso do desenvolvimento de um programa o programador faça uso de um tipo abstrato T cujas operações que o especificam são P_1, P_2, ... , P_m (Ex.: push e pop para pilhas). Os objetos desse tipo precisam ser implementados. Isso é feito através de um conjunto de variáveis V_1, V_2, ..., V_n , cujos tipos são diretamente, ou mais diretamente, representáveis na memória do computador, e de um conjunto de procedimentos, um para cada P_i, que operando sobre as variáveis V_j modelam as operações abstratas. O cluster para o tipo em questão teria a forma:

T = cluster < lista de parametros > is $P_1, P_2, \ldots, P_n$;
rep= < estrutura de representação para o tipo, definida com base em $V_1, V_2, \ldots V_n$ >

P_1 = proc .....; end P_1;

P_2 = proc .....; end P_2;

.
.
.
.

P_m = proc .....; end P_m ;

end T ;

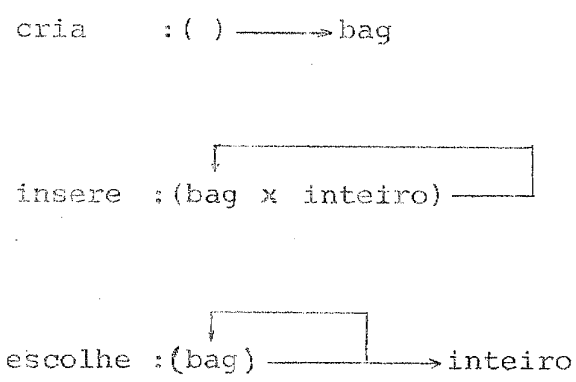
A presença de uma lista de parâmetros num cluster possibilita a definição de tipos mais genéricos. Isso é não somente útil como freqüente em programação. Um exemplo comum é a definição do tipo pilha; não existe muita diferença entre uma pilha de inteiros e uma pilha de reais. Assim sendo é preferível definir-se o tipo parametrizando-o, ou seja, pilha de t, onde t é um tipo pré-definido.

Associado a um cluster existem dois tipos: o tipo abstrato que está sendo definido (no caso, T) e sua representação indicado no rep. Fora do cluster os objetos do tipo T sempre serão tratos como tal, não existindo modo de se acessar seus constituintes (a não ser que exista uma operação explicitamente definida para isto). Dentro do cluster, contudo, os objetos do tipo T devem ser visto pelo tipo de sua representação, uma vez que as operações são definidas em termos dessa representação. Para permitir essa mudança de ponto de vista, a palavra reservada cvt aparece como um tipo dentro do cluster, podendo somente aí ser usada.

Um Exemplo

Suponha que queiramos modelar o conceito de um "bag" de inteiros. Um bag é semelhante a um conjunto, exceto pelo fa

to de em um bag um mesmo elemento poder aparecer mais de uma vez. As operações para esse tipo são aquelas para criar um bag vazio, para inserir um inteiro num bag, para escolher aleatoriamente um elemento no bag e retirá-lo, e para verificar se um bag é vazio. Esquematicamente:



Antes de passarmos a definição do cluster para o tipo em questão, precisamos escolher uma representação. Existem várias alternativas. Vamos adotar a seguinte: um registro composto de um vetor de 100 posições que conterá os elementos do bag e um inteiro para indicar o número de elementos no bag.

```
rep = record [a:array [1..100] of int, n:int]
```

O cluster para o tipo seria:

```
bag-inteiro = cluster is
    cria,      % cria um bag vazio
   insere,     % insere um elemento no bag
   escolhe,    % escolhe um elemento,aleatoriamente,
                 e o remove
   vàzia,      % inspeciona se um bag está vazio

  rep = record [a: array[1..100] of int,  n: int] ;

cria = proc () returns (cvt) ;
       return rep $ { a:[0,0,...,0], n:0 } ;
       end cria ;

insere = proc (x:cvt,  i:int) ;
       if x.n = 100 then erro
         else x.n:= x.n + 1 ;
              x.a[x.n]:=i ;
              end ;
       end insere;

escolhe = proc (x: cvt) returns (int);
       k,i:int ;
       if x.n=0 then erro
         else k:=  randint (1,x.n); % função aleatória
              i:=  x.a [k] ;
              x.a[k]:= x.a [x.n] ; % preenche o vazio
              x.n:= x.n - 1 ;
              end;
       return i;
       end escolhe;
```

```
        vazio = proc (x: cvt) returns (bool);
                if x.n = 0 then return true;
                           else return false ;
                end;
        end vazio;
end bag - inteiro;
```

Observações:

i) O procedimento escolhe faz uso de uma função randint, geradora de números aleatórios, entre os inteiros 1 e n. O número gerado durante uma chamada da função é então usado para escolha do elemento no"bag", caracterizando assim o fato de que para esse tipo de dados a busca de um elemento é por tentativas e erros.

ii) No procedimento cria a construção rep$\{.....\}$ é usada para criar uma instância nova do rep.

iii) A presença da palavra erro nos procedimentos insere e escolhe esta substituindo uma possível ação que necessitaria ser implementada.

iv) A representação do tipo bag-inteiro por meio de um vetor tem o inconveniente de limitar o seu tamanho. Isso seria contornado caso escolhêssemos uma representação por lista.

v) As operações de um tipo abstrato são invocadas por um nome composto do nome do tipo e do nome da ope

ração. Exemplo: Se b é do tipo bag-inteiro, bag-inteiro$insere(b,5) coloca 5 no "bag" b.

Estruturas de Controle

Uma estrutura de controle é uma construção que estabelece um método de sequenciamento de ações. Toda linguagem de programação provê o usuário com certas estruturas de controle, como,por exemplo, os comandos if e while.

A seleção dos objetos de uma coleção é uma tarefa rotineira em programação. Muitos loops se destinam unicamente a isto (Ex.: pesquisa de um elemento num vetor). Os comandos usados para seleção de objetos de uma coleção são, contudo, limitados a certos tipos (Ex.: o comando for de muitas linguagens itera apenas sobre os inteiros). CLU, assim como ALPHARD, provê um construtor linguístico chamado iterator,que possibilita ao usuário definir como os objetos de um tipo abstrato qualquer podem ser selecionados.

Um Exemplo

Considere que s seja uma variável do tipo string (um tipo primitivo em CLU). Duas operações definidas para esse tipo são size (s), que retorna um inteiro referente ao tamanho do string, e fetch (s,n) que devolve o n-ésimo caractere do string. Com essas operações podemos selecionar os caracteres de s um a um. Embora muito simples, vamos fazer isso através de um iterator:

```
cadeia = iter (s:string) yields (char) ;
         i:int:=1 ;
         limite: int:=string$size(s);
         while i <= limite do
               yield string$fetch(s,i);
               i:= i+1 ;
               end ;
       end cadeia ;
```

Esse iterator poderia ser usado num programa por meio do comando for. Como exemplo, suponha que quiséssemos saber o número de "a" que um string x contém. Escreveríamos o seguinte trecho de programa :

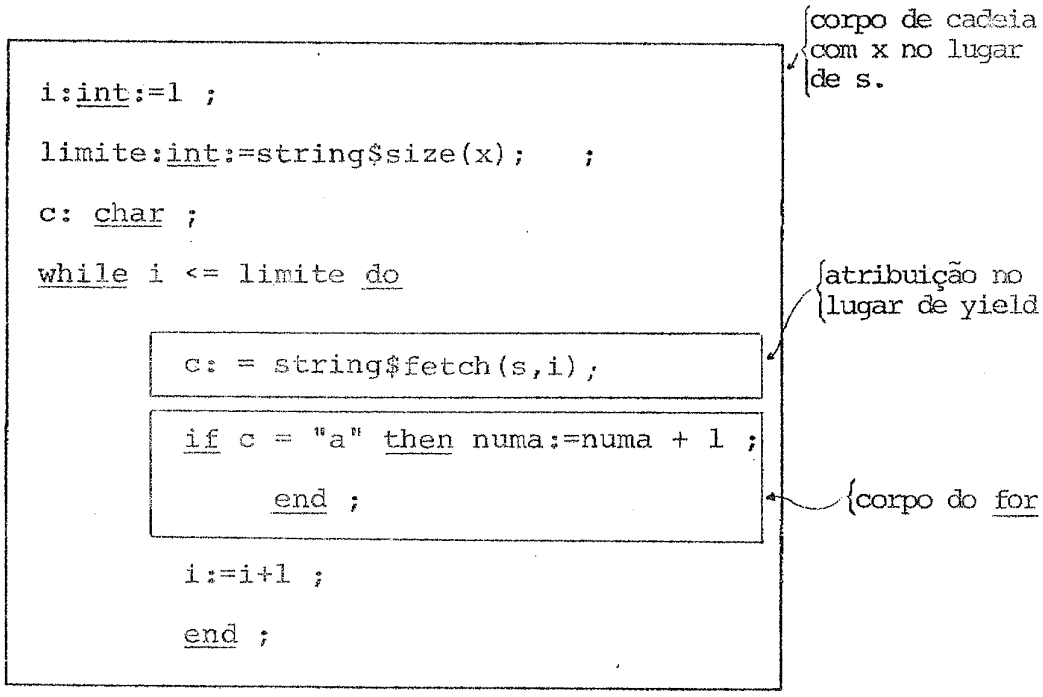
```
.............................
numa : int := 0 ;
for c : char in cadeia (x) do
   if c = "a" then numa: = numa + 1 ;
      end ;
end ;
.............................
```

A semântica desse construtor é a seguinte: Ao atingir o comando for,o iterator cadeia é invocado com seu argumento, iniciando-se assim sua execução. Quando o comando yield é executado o objeto produzido é então atribuído a variável declarada no comando for, dando assim início a execução do seu corpo. Após isto, o iterator é reassumido no comando que se segue ao yield. E assim sucessivamente, até o término da iteração, que

pode se dá pelo têrmino do iterator ou pelo têrmino prematuro do comando for .

O trecho de programa abaixo é equivalente ao anterior e ilustra exatamente o efeito do iterator:

```
.....................
numa: int := 0;
```



CAPÍTULO III

PROGRAMAÇÃO COM TIPOS ABSTRATOS DE DADOS

III.1 - Introdução

A estratégia de uso mais generalizado na construção de programas é, sem dúvida, o princípio de decomposição do problema em subproblemas, tendo como base o reconhecimento de abstrações. De acordo com essa estratégia um programa é construído em várias etapas. Na primeira etapa, a questão que se coloca é como implementar o comportamento abstrato requerido pelo problema como um todo. Tal implementação é então realizada pelo reconhecimento de um conjunto de abstrações (objetos e operações) adequados à solução do problema e elaboração de um algoritmo com base nessas. Isso feito, uma nova questão se coloca: como implementar as novas abstrações surgidas na etapa anterior. Cada uma delas é então implementada em termos de novas abstrações. Esse processo continua até que todas as abstrações introduzidas nas diversas etapas sejam implementadas com base naquelas presentes na linguagem em uso.

Esse método de desenvolver programas, ao separar nitidamente o uso de uma abstração de sua implementação, estimula o programador a deixar de lado a questão relativa a representação dos dados para se dedicar a elaboração de um algorítmo, expresso como um programa abstrato, operando sobre dados abstratos. Isso feito, o programador seleciona uma representação concreta para

os dados, em termos de tipos direta ou mais diretamente (abstratos, porém num nível de abstração menor) representáveis na memória do computador, programando as operações requeridas pelo programa abstrato.

São consequências naturais dessa metodologia:

i) Para ser compreendido a um nível abstrato, o comportamento dos dados abstratos deve ser completamente caracterizado pelo seu conjunto de operações.

ii) A fatoração da construção do programa em um programa abstrato e uma implementação do tipo ou tipos abstratos, alivia, além de tornar construtiva a tarefa de verificação de correção de programas , contribuindo de forma substancial para obtenção de programas confiáveis.

iii) A estrutura hierárquica e modular da construção do programa dá origem a programas que são razoavelmente fáceis de se entender, modificar e manter.

III.2 - Um Exemplo de Construção de Programa

Para mostrar como abstrações surgem naturalmente no curso do projeto de um programa, consideremos o seguinte problema:

Dado um texto, queremos computar o número de vezes que cada palavra ocorre no mesmo, bem como sua frequência, calculada como uma percentagem do número total de palavras do texto. O texto é representado como uma sequência de palavras e uma palavra é uma sequência de caracteres. Palavras adjacentes são separadas por um ou mais caracteres não alfabéticos, tais como espaços, pontos vírgulas, etc. Não existe diferença entre letras maiúsculas e minúsculas no reconhecimento de uma palavra.

A saída é uma sequência de caracteres, dividida em linhas. Cada linha contém uma palavra seguida do número de vezes de sua ocorrência e de sua frequência . As palavras são impressas em ordem alfabética. Por exemplo :

a	2	3.509%
acaso	1	1.754%
ana	1	1.754%
.	.	.
.	.	.

A solução que apresentamos a seguir faz uso da linguagem CLU e é uma adaptação daquela formulada por Liskov, Snyder, Atkinson e Schaffer em [25] .

Visto não termos qualquer informação sobre os meios físicos usados para conter o texto a ser processado (cartão, fi

ta magnética, etc.), bem como o resultado do processamento, consideremos os tipos abstratos entrada e saída. Assim sendo, o que especificamente queremos é escrever um procedimento para contar palavras, o qual recebe como argumentos dois objetos, um do tipo entrada e o outro do tipo saída, os quais são então alterados:

```
conta-palavras = proc (e:entrada, s:saída) ;
    ...
    end conta-palavras;
```

Entre as operações para o tipo entrada, aquelas que mais nos interessa são próximo(e) , que extrai e devolve o primeiro caractere da sequência e, e vazio(e) , que devolve true caso não exista mais caracteres em e e false em caso contrário. Para o tipo saida, a operação mais interessante é escreve-seq(x,s) que acresce o string x à sequência de caracteres existente em s.

Feitas essas considerações, passemos à implementação de conta-palavras. O comportamento abstrato desse procedimento parece nos induzir as seguintes observações:

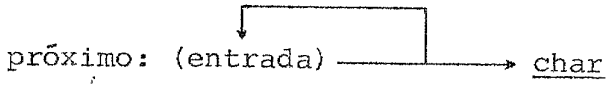
i) O procedimento deve selecionar cada palavra do texto. Isso significa que devemos escolher uma representação para os objetos palavras e saber como cada palavra é obtida. Para representar palavras optamos pelo tipo string que é primitivo em CLU. Quanto a sua obtenção do texto, definimos um procedimento próxima-palavra(e) , que recebe o objeto

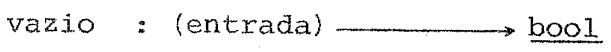
e do tipo entrada e devolve a primeira palavra de e ou o string vazio (" "), caso e seja vazio.

ii) O procedimento necessita esquadrinhar todo o texto antes de iniciar a impressão do resultado. Isso significa que de alguma forma é necessário reter as informações sobre cada palavra entre essas ações. Vamos fazer isto definindo o tipo abstrato bag-de-palavras, cujas operações são cria(), que cria um bag-de-palavras vazio, insere(bp,p), que insere a palavra p no bag-de-palavras bp, e escreve(bp,s), que escreve o bag-de-palavras em s que é do tipo saída.

Esquematicamente, faremos uso dos seguintes tipos abstratos e respectivas operações:

1. Entrada





2. Saída

escreve-seq: (string, saída)

3. Bag-de-Palavras

cria : ( ) ⟶ bag-de-palavras

insere: (bag-de-palavras, string)

escreve: (bag-de-palavras, saída)

Os tipos bool, char e string são primitivos em CLU. O tipo char é usado para representar caracteres, enquanto o tipo string é usado na representação dos objetos abstratos palavras, visto que aquele contém as operações necessárias para este (" " palavra-nula e append - concatenação).

Podemos, agora, implementar conta-palavras conforme segue:

```
conta-palavras = proc (e: entrada, s: saida) ;

    % cria um bag-de-palavras vazio
    bp:bag-de-palavras := bag-de-palavras$cria( );
```

```
        % esquadrinha o texto, colocando em bp cada pala
          vra encontrada
        p: string := próxima-palavra(e)   ;
        while p ~= " " do
            bag-de-palavras$insere(bp,p)
            p:= próxima-palavra(e)   ;
            end ;

        % impressão do bag-de-palavras
           bag-de-palavras$escreve(bp,s);

    end conta-palavras ;
```

Relembramos o seguinte, com relação a linguagem CLU:

i) % introduz comentários.

ii) A notação variável:tipo é usada para declarar variáveis.

iii) As operações sobre dados abstratos são invocadas na seguinte forma: nome do tipo, seguido de $, guido do nome da operação, seguido dos parâmetros da operação.

O procedimento próxima-palavra é o seguinte:

```
próxima-palavra = proc (e:entrada) returns(string);

   c:char :=" " ; % caractere branco(não alfabético)

   % procura o primeiro caractere da palavra
   while ¬alfab(c) do
       if entrada$vazio(e)
          then return " " ;
          end ;
       c:= entrada$próximo(e);
       end;

   % obtém a palavra por concatenação de caracteres
   p:string:=" " ; % string vazio (palavra nula)
   while alfab(c) do
          c:= minuscula(c) ;
          p:= string$append(p,c); % concatenação
          if  entrada$vazia(e)
                    then return p;
                    end ;
          c:= entrada$proximo(e)
          end ;
   return p; % o último caractere não-alfabético
             é desprezado

end próxima-palavra;
```

Observe a introdução de dois procedimentos adicionais: alfab(c) que testa se o caractere c é alfabético, e minuscula(c)

que retorna a letra minúscula correspondente ao caractere c. Não implementaremos esses procedimentos.

Precisamos agora implementar o tipo abstrato bag-de-palavras, cujas operações, conforme já dissemos e fizemos uso delas na implementação de conta-palavras, são:

cria : cria um bag-de-palavras vazio

insere(bp,p) : insere a palavra p no bag-de-palavras bp.

escreve(bp,s) : escreve o bag-de-palavras bp em s que é do tipo saída.

Antes de mais nada, precisamos escolher uma representação para o tipo bag-de-palavras. Contudo, essa representação deve ser tal que propicie um armazenamento razoavelmente eficiente para as palavras, bem como facilite a impressão das mesmas em ordem alfabética, juntamente com suas estatísticas. O cálculo das estatísticas, com certeza, será facilitado se mantivermos um contador para o número de palavras inseridas no bag-de-palavras. Uma vez que é esperado que o número total de palavras do texto seja bem maior que o número de palavras distintas, é conveniente que cada palavra distinta seja representada por um único item, o qual mantém um contador para o número de ocorrências daquela palavra. Uma consequência imediata disso é que a inserção de uma palavra tem que ser precedida de uma pesquisa para se saber se a palavra já se encontra no bag-de-palavras. A representação deve facilitar essa busca assim como a inserção. Uma estrutura ideal para essa representação parece ser a árvore binária.

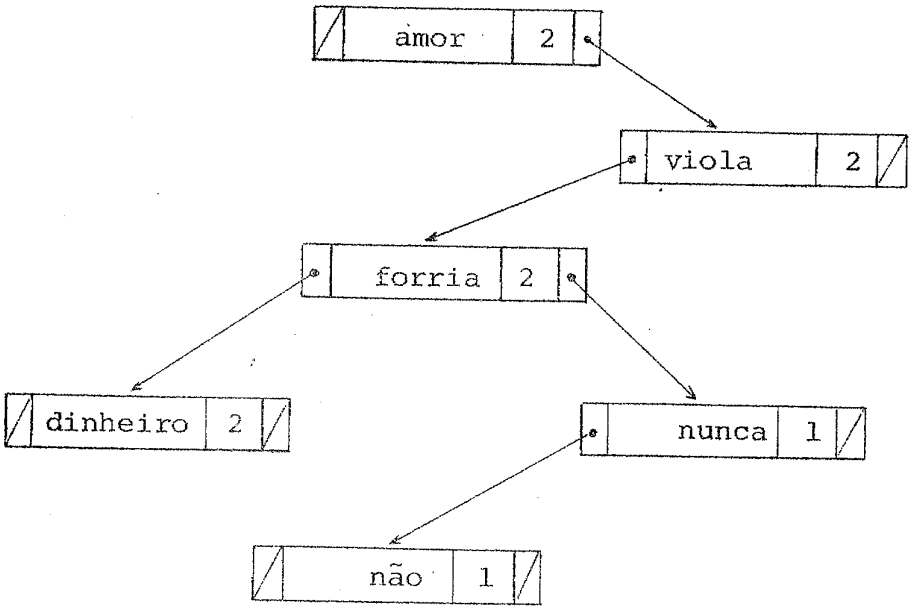
Vamos escolher então para o bag-de-palavras a seguinte representação: um registro com dois campos, um para conter as palavras e que será um objeto de tipo árvore, e o outro para conter o número total de palavras do bag-de-palavras, este do tipo inteiro:

rep=record [conteúdo:árvore, total: int]

Como exemplo, considere o seguinte texto:

"Amor, viola, forria, nunca dinheiro.
Viola, forria, amor, dinheiro não".

O bag-de-palavras teria total = 10 e conteúdo o valor indicado na árvore abaixo:



As operações para o tipo abstrato árvore são:

cria ( ) : cria uma árvore vazia

insere (a,p) : devolve uma nova árvore que resulta da inserção da palavra p na árvore a

em-ordem(a) : devolve os nós da árvore a segundo a ordem alfabética das palavras na árvore (um iterator).

A implementação do tipo bag-de-palavras é feita através do seguinte cluster:

```
bag-de-palavras = cluster is
            cria ,     % cria um bag-de-palavras vazio
          insere ,     % insere uma palavra num bag-
                         -de-palavras
          escreve;     % imprime o conteúdo de um bag-
                         -de-palavras

    rep=record [conteúdo: árvore, total: int];

    cria = proc ( ) returns (cvt) ;
           return rep${ conteúdo:árvore$cria(),to-
                        tal:0};
           end cria;

  insere = proc(bp:cvt, p:string) ;
           bp.conteúdo :=árvore$insere(bp.conteúdo,
                        p);
           bp.total := bp.total + 1;
           end insere ;
```

```
        escreve = proc(bp:cvt, s:saida) ;
                  for valor: string, frequencia: int
                      in arvore$em-ordem(bp.conteúdo)do
                         imprima-linha(valor,freqüência,
                         bp.total, s);
                         end;
                  end escreve;

    end bag-de-palavras;
```

Observamos que:

i) cvt é usado para que um objeto do tipo definido pelo cluster seja visto, dentro deste, como do tipo indicado no rep.

ii) O construtor rep${...} é usado para criar uma instância do objeto indicado no rep.

iii) A notação variável.campo (bp.conteúdo, em insere) é usada para seleção de um campo de um objeto que seja do tipo record.

iv) No procedimento escreve o comando for faz uso do iterator em-ordem, a ser implementado com o tipo árvore, que a cada iteração atribui à variável valor uma palavra e à variável freqüência o número de ocorrências dessa palavra.

v) Um novo procedimento foi introduzido na implementação de escreve, imprima-linha, o qual recebe uma palavra, sua frequência, o total de palavras do bag-de-palavras e um objeto do tipo saida, s, produzindo neste a impressão de uma linha. Omitiremos a implementação desse procedimento.

Vamos tratar agora da implementação do tipo árvore. Começaremos pela representação dos nós.

Conforme já ficou evidenciado, um nó para esta árvore apresenta quatro campos: um para reter a palavra, outro para conter a frequência da palavra, e os outros dois para indicar as subárvores da esquerda e da direita, as quais, respectivamente, precede e sucede alfabeticamente a palavra contida no nó. Um nó é, assim, um registro:

```
nó = record [valor: string,freqüência: int ,
             esquerda:árvore,direita:árvore ]
```

Uma árvore é, agora, recursivamente definida como um objeto do tipo nó ou uma árvore vazia. A árvore vazia será representada por um objeto do tipo null (primitivo em CLU) cujo único valor é nil. Assim, a representação do tipo árvore é uma união descriminada.

```
rep = oneof [vazio: null, nao-vazio: nó]
```

Podemos agora escrever o cluster para o tipo árvore:

```
árvore = cluster is
         cria        % cria uma árvore vazia
         insere      % acresce uma palavra à árvore
         em-ordem    % atravessa a árvore  em ordem

         nó = record [valor:string,freqüência:int ,
                      esquerda:árvore,direita:árvore]
         rep = oneof[ vazio: null, nao-vazio: nó] ;

      cria = proc ( ) returns (cvt);
             return rep$ {make-vazio(nil)};
             end cria ;

      insere=proc(a:cvt, p:string) returns (cvt);
             tagcase a
                tag vazio:
                   n:nó:= nó$ { valor:p, freqüência:1,
                                esquerda:árvore$cria(),
                                direita :árvore$cria()};
                   return rep${make-nao-vazio(n)};

                tag nao-vazio(n:no):% atribui a à variá
                                      vel n
                  if p = n.valor
                     then n.freqüência:=n.freqüência+1 ;
                   elseif p < n.valor
                     then n.esquerda:=árvore$insere (n.es
                                querda, p);
```

```
                    else n.direita:=arvore$insere(n.di
                                        reita,p);

                  end;
                return a ;
              end ;
            end insere ;

  em-ordem=iter(a:cvt)yields (string, int)
          tagcase a
            tag vazio:; % nada a produzir
            tag nao-vazio(n:no): % atribui a â variável n
              for valor: string, freqüência: int;
                 in arvore$em-ordem(n.esquerda)do
                    yield valor, freqüência ;
                    end ;
              yield  n.valor, n.freqüência ;
              for   valor:  string, freqüência: int
                    in arvore$em-ordem(n.direita)do
                       yield valor, freqüência
                       end ;
          end ;

        end em-ordem;

end  árvore;
```

Observações:

i) Os procedimentos foram definidos recursivamente.

ii) Objetos do tipo oneof [$s_1:T_1$, $s_2:T_2$,...,$s_n:T_n$] são criados pela operação make-s_i(x), com x sendo do tipo T_i

iii) O tipo de uma instância de um objeto y do tipo oneof é detectado pelo comando tagcase y. Se y for do tipo T_i o programa segue a alternativa tag s_i

III.3 - Seleção da Representação

A distinção, bem nítida, entre abstração e implementação facilita a modificação e manutenção de programas. Uma vez que tenha sido detectado a necessidade de reimplementação de uma abstração, os módulos que usam a abstração não são de forma alguma alterados. Somente o cluster que implementa a abstração precisa ser examinado. Uma consequência saudável desse fato, é que o usuário pode, deliberadamente implementar um tipo de diversas formas e ver na prática, conforme a natureza da aplicação, qual seja a melhor implementação para o tipo.

Considerando o exemplo da seção anterior, observamos , que o procedimento conta-palavras faz uso das abstrações entrada, saída e bag-de-palavras . O procedimento não depende de como esses objetos são implementados, mas apenas do comportamento das operações definidas para esses objetos. Desse modo, qualquer mo

dificação na representação dos objetos daqueles tipos é completamente transparente ao usuário de conta-palavra.

CAPÍTULO IV

OUTRAS CONSIDERAÇÕES

Abstração é um processo inerente às aplicações dos computadores ao mundo real. Em particular, tipos abstratos de dados têm se revelado uma importante ferramenta no desenvolvimento de programas. Seu uso permite ao programador escrever programas em termos dos tipos de dados que ele considera melhor se adequarem à solução de um dado problema, ao invés de se restringir ao elenco de tipos previamente definidos e disponíveis na linguagem de programação usada para codificar o programa. O resultado é a fatoração do programa em duas partes: uma parte abstrata, que manipula os objetos dos tipos abstratos de dados por meio de suas operações, e uma parte de implementação, que representa os objetos e operações abstratas por meio de outros mais concretos [22]. O emprego, talvez iterado, dessa metodologia dá ao programa uma estrutura elegante e permite a fatoração natural de várias tarefas de programação: especificação, desenvolvimento, documentação, verificação, testes, etc.

Para realmente se tirar proveito da abstração, é necessário que os tipos de dados sejam especificados de maneira formal e independente de qualquer particular representação [24].

A linguagem usada para se especificar um tipo abstrato é de fundamental importância. Os tipos abstratos de que lançamos mão para o desenvolvimento do exemplo do capítulo anterior tive -

ram o comportamento de suas operações definido de maneira totalmente informal. Contudo o uso de uma linguagem informal, tal como a linguagem natural, nem sempre se mostra eficiente para criação e comunicação de abstrações. Convém salientar que somente com muito cuidado e atenção é possível escrever uma especificação precisa em linguagem natural. O problema é mais grave ainda quando a abstração tem que ser comunicada a alguém (lembre-se de que o especificador de um tipo abstrato nem sempre é o seu implementador ou usuário). A especificação pode não somente ser imprecisa e portanto ambígua, como a própria linguagem na qual a especificação é feita conter ambigüidades. Se a ambigüidade é percebida, esta pode ser resolvida. Porém, o que normalmente ocorre é cada um dos envolvidos com a abstração formar sua própria concepção da abstração, criando assim um sério problema de interface.

Naturalmente o uso de uma linguagem formal não nos garante uma especificação livre de ambigüidades ou inconsistências. O que esta nos propicia são critérios para o reconhecimento de ambigüidades e inconsistências, aumentando assim as possibilidades de detecção de falhas na especificação. A garantia de que a especificação é correta é primordial uma vez que é contra esta que se tem que testar a implementação do tipo abstrato.

Vários métodos têm sido propostos para especificação de um tipo abstrato de dados (Liskov e Zilles [26] Dentre estes o método algébrico tem recebido muita atenção, sendo apresentado em vários trabalhos (Zilles [38], Goguen-Thatcher-Wagner [10] e Guttag-Horning [14]) como adequado para especificação de um tipo abstrato. Segundo essa abordagem uma especificação para um tipo T consiste de uma descrição sintática e uma descrição semântica;

a especificação sintática define os nomes, domínios e codomínios das operações de T, enquanto a especificação semântica contêm um conjunto de axiomas, na forma de equações, relacionando as operações do tipo T umas com as outras.

No processo de desenvolvimento de programas que manipulam tipos abstratos de dados é de fundamental importância o conceito de implementação. Uma implementação consiste basicamente na representação de um tipo de dados em outro. O segundo tipo é considerado um tipo de dados mais concreto que o anterior, no sentido de que ele está, de alguma forma, mais próximo dos tipos de dados da linguagem de programação na qual o programa será expresso. Essa etapa repete-se sucessivamente até que o tipo de dados original seja inteiramente expresso em termos dos tipos de dados da linguagem de programação em apreço.

A noção de implementação tem sido objeto de frequentes discussões e vários esforços foram feitos no sentido de precisar-lhe o significado. Dentre os trabalhos com esse objetivo duas abordagens do problema têm merecido destaque: uma considera a implementação como uma função abstrata definida do espaço concreto no espaço abstrato; a outra considera uma função de representação que associa a cada termo do tipo a ser implementado um outro do tipo usado na implementação.

A primeira abordagem foi apresentada inicialmente por Hoare em [15] e se constitui numa forma mais natural se o tipo de dados é visto como classes de equivalência dos termos, uma vez que um mesmo objeto abstrato pode ter várias representações.

A outra abordagem é brevemente apresentada por Goguen , Thatcher e Wagner em [10] e de modo mais completo e formal por

Ehrig, Kreowski e Padawitz em [05] e [06] . Tal abordagem parece ser mais conveniente se alguém está interessado em "sistemas de reescrita ou problemas de tradução" (Gaudel [09]).

Uma abordagem mais recente do problema da implementação é devido a Pequeno ([30]). Utilizando os recursos da lógica matemática ele descreve formalmente o processo de implementação como uma interpretação entre teorias.

Os aspectos relativos à especificação e implementação de tipos abstratos de dados são abordados em maiores detalhes em [32].

REFERÊNCIAS BIBLIOGRÁFICAS

[01] BRAND, D., "A Note on Data Abstractions", SIGPLAN Notices, Vol. 13, nº 1, 1978, pp.21-24 .

[02] CARVALHO, R.L., Maibaum, T.S.E., Pequeno, T.H.C., Borquez, A.A.P. e Veloso, P.A.S., "A Model Theoretic Approach to the Theory of Abstract Data Types and Data Structures", Research Report CS-80-22, University of Waterloo, Ontario, Canadá, 1980.

[03] DAHL, O.J. e Hoare, C.A.R., "Hierarchical Program Structures", em STRUCTURED PROGRAMMING, Academic Press, London e New York, 1972, pp. 175-220.

[04] DENNIS, J., "An Example of Programming With Abstract Data Types", SIGPLAN Notices, Vol. 10, nº 7, 1975, pp. 25-29.

[05] EHRIG, H., Kreowski, H.-J.e Padawitz, P., "Some Remarks Concerning Correct Specification and Implementation of Abstract Data Types", Informatik Research Report nº 77-13, Technical University Berlin, 1977.

[06] EHRIG, H., Kreowski, H.-J.e Padawitz, P., "Stepwise Specification and Inplementation of Abstract Data Types", Informatik, Technical University Berlin, 1978.

[07] FLON, L., "Program Design With Abstract Data Types", Department of Computer Science, Carnegie-Mellon University , Pittsburgh, Pa., 1975

[08] FLON, L., "A Survey of Some Issues Concerning Abstract Data Types", Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., 1974.

[09] GAUDEL, M.C., "Algebraic Specification of Abstract Data Types", IRIA Rapport de Recherche nº 360, Roqquencourt , 1979.

[10] GOGUEN, J.A., Thatcher, J.W. e Wagner, E.G., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", em CURRENT TRENDS IN PROGRAMMING METHODOLOGY, Vol. IV, Prentice - Hall, 1978, pp. 81-149.

[11] GRIES, D. Gehani, N., "Some Ideas on Data Types in High-Level Languages", Comm. ACM, Vol. 20, nº 6, 1977, pp. 414-420.

[12] GUTTAG, J., "Abstract Data Types and the Development of Data Structures", CACM, Vol. 20, nº 6, 1977, pp. 396-404.

[13] GUTTAG, J., Horowitz, E. e Musser, D., "Abstract Data Types and Software Validation", Comm. ACM, Vol. 21, nº 12, 1978, pp. 1048-1064.

[14] GUTTAG, J. e Horning, J.J., "The Algebraic Specification of Abstract Data Types", Acta Informatica, Vol. 10, nº 1 , 1978, pp. 27-52.

[15] HOARE, C.A.R., "Proof of Correctness of Data Representations", Acta Informatica 1, 1972, pp. 271-281.

[16] HOARE, C.A.R., "Proof of a Structured Program: The Sieve of Eratosthenes", The Computer Journal, Vol. 15, nº 4, 19, pp. 321-325.

[17] HOARE, C.A.R., "Notes on Data Structuring", em STRUCTURED PROGRAMMING, Academic Press, London e New York, 1972 , pp. 83-174.

[18] HOARE, C.A.R., "Data Structures", em CURRENTS TRENDS IN PROGRAMMING METHODOLOGY, Vol. IV, Prentice-Hall, 1978.

[19] JENSEN, K. e Wirth, N., PASCAL: USER MANUAL AND REPORT , Springer-Verlag, New York, 1975.

[20] LEDGARD, H., "ADA : AN INTRODUCTION", Springer-Verlag, New York, 1981.

[21] LINDSEY, C.H., Meulen, S.G., INFORMAL INTRODUCTION TO ALGOL 68, North-Holland Publishing Company, New York, 1977.

[22] LISKOV, B e Zilles, S., "Programming with Abstract Data Types", Proccedings of a Symposium on Very High Level Languages, SIGPLAN Notices, Vol. 9, nº 4, 1974, pp. 50-59.

[23] LISKOV, B., Zilles, S., "An Approach to Abstraction", Computation Structures Group Memo 88, MIT, Cambridge, Massachusetts, 1973.

[24] LISKOV, B., "Data Types and Program Correctness", SIGPLAN Notices, Vol. 10, nº 7, 1975, pp. 16-17

[25] LISKOV, B., Synder, A., Atkinson, R., Schaffer, C., "Abstraction Mechanisms in CLU", Comm. ACM, Vol. 20, nº 8 , 1977, pp. 564-576

[26] LISKOV, B. e Zilles, S. "An Introduction to Formal Specifications of Data Abstractions", em CURRENT TRENDS IN PROGRAMMING METHODOLOGY, Vol. I, Prentice-Hall, 1978

[27] LUCENA, C.J.P. e Pequeno, T.H.C., "Program Derivation Using Data Types: A Case Study", IEEE Transations on Software Engineering, Vol. Se-5, nº 6, 1979, pp. 586-592

[28] LUCENA, C.J.P. , ANALISE E SÍNTESE DE PROGRAMAS: UMA INTRODUÇÃO, Segunda Escola de Computação, Campinas, 1981.

[29] MANNA, Z., MATHEMATICAL THEORY OF COMPUTATION, McGraw-Hill Book Company, New York, 1974.

[30] PEQUENO, T.H.C., "Uma Descrição Formal dos Processos de Especificação e Implementação de Tipos Abstratos de Dados", Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Tese de Doutorado, 1981.

[31] PEREDA, A.A., "Métodos de Descrição de Tipos de Dados Estruturas de Dados", Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro, Tese de Doutorado, 1979.

[32] PESSOA, F.E.P e Veloso, P.A.S., "Introdução à Especificação e Implementação de Tipos Abstratos de Dados, Monografias em Ciência da Computação, Departamento de Informática , Pontifícia Universidade Católica do Rio de Janeiro, a ser publicado.

[33] SHAW, M., Wulf, W. e London, R., "Abstraction and Verification in ALPHARD: Iteration and Generators", Technical Report, Carnegie-Mellon University, Pittsburgh, Pa. e USC Information Sciences Institute, Marina Del Rey, California, 1976.

[34] VELOSO, P.A.S. e Pequeno, T.H.C., "Don't Write More Axioms Than You Have to: A Methodology for the Complete and Correct Specification of Abstract Data Types; With Examples", Monografias em Ciência da Computação, nº 10, Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

[35] WIRTH, N., PROGRAMAÇÃO SISTEMÁTICA, Editora Campus Ltda. , 1978.

[36] WULF, W., London, R. e Shaw, M., "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology", Technical Report, Carnègie-Mellon University, Pittsburgh, Pa. and USC Information Sciences Institute , Marina Del Rey, California, 1976

[37] WULF, W., London, R. e Shaw, M., "An Introduction to the Construction and Verification of ALPHARD Programs" , IEEE Transactions on Software Engineering, Vol. Se-2 , nº 4, 1976, pp. 253-265

[38] ZILLES, S., "Algebraic Specification of Data Types", Computation Structures Group Memo 119, MIT, 1975