5. Abstração I: Encapsulamento
Pratt & Zelkowitz

Recursos para programar grandes sistemas

- 5.1 Tipos abstratos de dados.
- 5.2 Encapsulamento por subprograma.
- 5.3 Definição de tipos.
- 5.4 Administração de armazenagem.

Meta das LP

Diferenças entre as várias formas de dados ⇒ transparentes para o programador que os usa.

- desnecessário preocupar com a implementação.
- ortogonalidade: mesma sintaxe em declarações e semântica comparável em assinaturas.

add: inteiro x inteiro → inteiro

matricular-aluno: Taluno x Tturma → Tturma

```
type Tturma = record
        disc: string[10];  id: char; vagas: integer;
        aluno: array [1..mxT] of  Taluno;
end;
```

Mecanismos para Criar Novos Tipos e Operações sobre eles

- Subprogramas:
  - criados pelo programador.
  - realizam as funcionalidades do novo tipo de dados que o subprograma representa.
  - responsabilidade do uso correto é do programador.
  - há pouco suporte automatizado da LP para prevenir uso inadequado.
- Declarações de tipo
- Herança.

Mecanismos
Declarações de tipo

- LP tem recursos para declaração de novos tipos e operações sobre o tipo.
- Conceito de tipo abstrato de dados é disponível em C, Pascal, Java, Ada, etc.:
  - a) pela criação de novos tipos pelo programador.
  - b) LP dá suporte para detectar o uso inválido dos novos tipos.

Mecanismos

Herança

- O conceito de POO e herança facilitam:
  - a) ao programador criar novos tipos de dados e operações sobre eles.
  - b) à LP dar o suporte para detectar o uso impróprio dos tipos criados.
- Facilita a criação de novos tipos com o conceito de classes e heranças.

Tipos Abstratos de Dados

- Fortran e Cobol limitam a criação de novos tipos de dados à definição de subprogramas.
- Abordagem atual é as LP terem facilidades para especificar e implementar TDA.
  - TDA podem ser implementados em Ada com package e como classes em C++ ou Java.
- Nesse capítulo é abordada a criação de TDA por meio de subprogramas.

Tipos Abstratos de Dados

Evolução do conceito de TD

- Início: aplicável a uma variável individualmente:
  - conjunto de valores que *uma* variável poderia assumir.*
- Pascal: aplicável a um conjunto de variáveis:
  - define a estrutura de um OD e suas ligações.
- Atualmente: aplicável a um conjunto de OD e ao conjunto das operações para manipulá-lo.
  - Os detalhes da implementação (RA) dos objetos são encapsulados (escondidos do programador).
    - visão: nome do tipo e lista de operações disponíveis

Tipo Abstrato de Dados

Definição

1. conjunto de OD, usando uma ou mais definição de tipos;
2. conjunto de operações abstratas sobre aqueles OD; e
3. encapsulamento do todo, criando um novo tipo, de forma tal que o usuário não pode manipulá-lo a não ser via as operações definidas.

Tipos de Dados Abstratos

Informação escondida

- Programação de grandes programas exige modularização (dividir para conquistar).
  - cada módulo com funcionalidade específica
    - executa um conjunto limitado de operações sobre uma quantia limitada de dados.
    - facilita o entendimento do projeto.
- Enfoques para modularização:
  - decomposição funcional (comuns nos 60's)
    - módulo: subprograma, função ou procedimento.
  - decomposição de dados: módulo é a abstração

Tipos de Dados Abstratos

Informação escondida

- Na decomposição funcional o programador deve conhecer detalhes dos OD e das operações.
- Na decomposição de dados é necessário saber *apenas* a especificação dos tipos de dados e as operações disponíveis.
- Métodos como refinamento passo a passo, programação modular, programação topdown são todos referentes a abstração.

Informação Escondida

Encapsulamento

- Com informação encapsulada em abstração:
  a) o usuário não necessita conhecer informações ocultas para usar a abstração.
  b) o usuário não pode usar a informação oculta de forma direta, mesmo se desejar.
- Agrupa componentes do programa em unidades ⇒ facilita o uso pelo usuário.
- É uma construção lingüística que suporta módulos com informações escondidas.

Informação Escondida

Encapsulamento

- Descreve módulos em 2 partes: interface e implementação.
  - Interface:
    - quais serviços são providos pelo módulo.
    - como eles podem ser acessados.
    - conjunto de entidades exportadas pelo módulo
      os módulos clientes importam essas entidades.
  - Implementação:
    - descreve os detalhes internos do módulo.
    - codifica os algoritmos que implementam as operações.

Informação Escondida

Encapsulamento

- Subprograma é um mecanismo básico de encapsulamento, existente em muitas LP.
  - permite fácil modificação do programa.
- Informação escondida
  - uma questão de projeto do programa.
- Encapsulamento
  - uma questão de projeto de LP, pois cabe a LP impedir o acesso as informações escondidas.

Abstração I: Encapsulamento

Encapsulamento por subprogramas

- Um subprograma é uma operação abstrata definida pelo *programador*.
- Hugs:

```
fat :: Integer –> Integer
fat n = product [1..n]
```

- Pascal:

```
function fat(n:integer):integer;
   begin
     if (n=0) or (n=1) then fat:=1
     else if n>1 then fat:=n*fat(n-1);
   end ;
```

Encapsulamento p/ Subprograma

Subprograma como operação abstrata

- Definição de subprograma é em duas partes: especificação e implementação:
- 1. especificação:
  a) nome do subprograma e dos argumentos (parâmetros formais)
  b) assinatura (ou protótipo),
  c) ação realizada.

  Subprograma representa uma função matemática
  S: argumentos → resultados

Subprograma como OA

Especificação de subprograma

- Subprograma f é uma função se retorna um resultado único e elementar:

  assinatura: f: $T_1 x\ T_2 x ... x\ T_n \rightarrow R$

  a) especificação em Pascal

  function f(x_1:T_1, x_2:T_2,...,x_n:T_n):R

  A palavra chave function é utilizada em Pascal e Fortran.

  b) especificação em C

  float f (**float** X, **int** Y);

Subprograma como OA

Especificação de subprograma

- Subprograma p é um procedimento se retorna resultados múltiplos ou estruturados
  - a) p é procedure (em Pascal, Fortran e Ada)
  - b) assinatura p: $T_1 x\ T_2\ x ... x\ T_n \rightarrow R_1 x\ R_2 x ... x\ R_k$
    - especificação em C
      void p ($T_1\ x_1$, ..., $T_n\ x_n$, $R_1\ *y_1$, ..., $R_k\ *y_k$);
    - especificação em Ada
      procedure p (X: **in real**; Y: **in integer**; Z: **in out real**; W: **out Boolean**)
      p: real x integer x real → real x Boolean

Subprograma como OA

Especificação de subprograma

- Especificação precisa da função computada é problemática:
  - a) podem existir argumentos implícitos,
  - b) resultados implícitos (efeitos colaterais)
  - c) subprograma não definido p/ algum argumento
    - para algum valor, transfere o controle para outro subprograma para tratar exceções/término anormal.
  - d) subprograma sensível à história
    - retém valores entre chamadas.
    - depende de tudo que já foi informada até agora.

Subprograma como OA

Implementação de subprograma

- Subprograma representa parte da VM construída pelo programador:
  - é implementado usando a estrutura de dados e as operações providas pela LP.
- Implementação

Encapsulado
- cabeçalho → assinatura
- declarações → locais
- comandos → definem a ação

  - uso: apenas via chamada ao subprograma.

Subprograma como OA

Implementação de subprograma

```
function qsort(p:ponteiro):ponteiro;
    px,py: ponteiro;
    procedure divide(var p,px,py:ponteiro);
begin
   if  (p=nil) or  (p^.prox=nil) then qsort :=p
   else begin
     divide (p,px,py);
     qsort:=concatena (qsort (px), concatena
                (p,qsort(py)))
   end;
end;
```

Subprograma como OA

Implementação de subprograma

- Subprogramas locais:
  - são subprogramas definidos aninhados, dentro de outros subprogramas (Pascal e Ada).
    - não são acessíveis externamente; estão encapsulados
- Chamada de Subprogramas
  - invocação checa os tipos dos argumentos,
  - checagem é estática ou dinâmica,
  - pode ocorrer coerção automática dos argumentos.

Encapsulamento p/ Subprograma

Definição e invocação de subprograma

- Definição
  - é a forma (estática) escrita do subprograma,
  - é a única informação disponível em tempo de tradução (só o tipo das variáveis é conhecido).
- Ativação
  - existe somente durante a execução,
  - l-value e r-value podem ser acessados, mas os tipos dos OD podem não estar disponíveis,
  - usa a definição como gabarito, de modo similar ao tipo de dados (para definir tamanho e RA)

Encapsulamento p/ Subprograma

Definição e invocação de subprograma

- A definição de subprograma e sua área de ativação: analogia com tipo e OD daquele tipo.
- No programa em execução:
  - cria-se uma área de ativação a cada chamada,
  - quando o subprograma completa sua execução, a área de ativação é destruída,
  - de uma única *definição* de subprograma, muitas áreas de ativações podem ser criadas,

Encapsulamento p/ Subprograma

Definição e invocação de subprograma

- A tradução da definição do subprograma permite dimensionar as RA para os OD e código executável, em tempo de tradução.
  - esse resultado é usado como um gabarito para as ativações do subprograma.
  - esse gabarito é divido em duas partes:
    - segmento de código
    - registro de ativação

Encapsulamento p/ Subprograma

Definição e invocação de subprograma

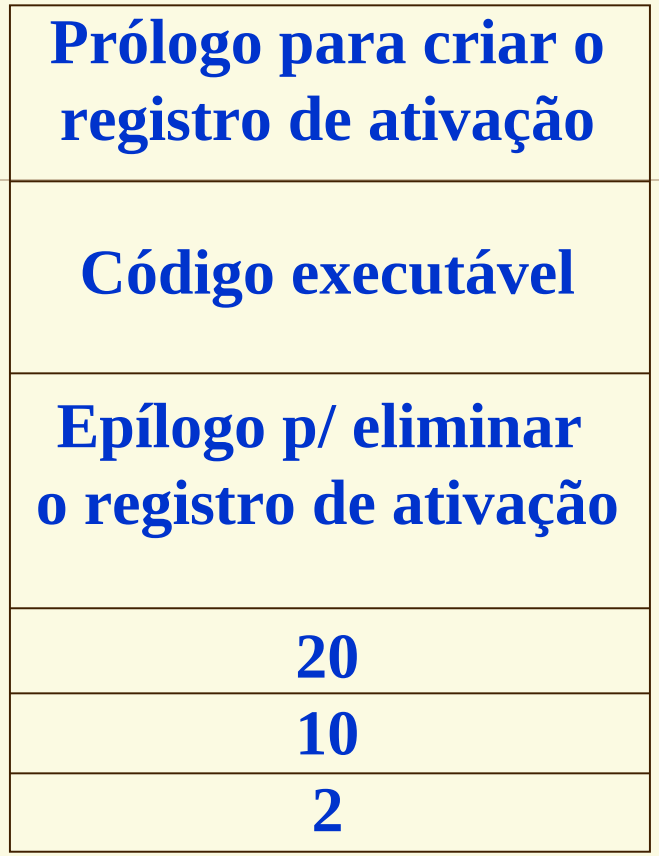
- Segmento de código:
  - parte estática, *invariante* durante a execução: consiste de constantes e do código executável.
  - cópia única é *cotizada* por todas as ativações.
- Registro de Ativação
  - parte dinâmica que contém as informações que variam durante a execução: resultado das funções, parâmetros, variáveis locais e estruturas para housekeeping (dados não locais, temporários, pontos de retorno, etc.).

Encapsulamento p/ Subprograma

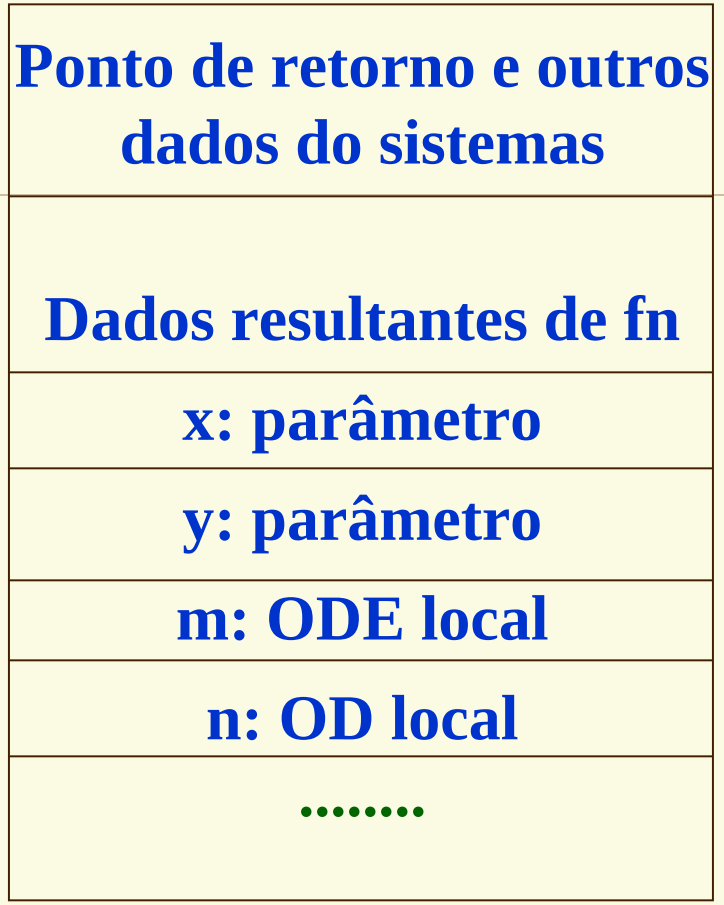
Definição e invocação de subprograma

```
float fn (float x, int y)
  { const int vi = 2;
    #define vf 10
     float m[vf]; int n;
.....
n = vi;
if (n < vi) { ... };
return (20*x+m[n]);}
```

1. fn: float x int → float
Dá as informações p/ RA dos parâmetros e do resultado.
2. Declarações de m e n induz a RA deles.
3. RA para literais e constantes: vi é constante, vf é definida como constante 10 (macro); 10 e 20 são literais.
4. RA para armazenar o código executável criado a partir dos comandos.



Segmento de código para a função fn



Registro de ativação p/ fn (gabarito)

Um por chamada

Definição e invocação de subprograma

ODE registro de ativação

- Tamanho: determinado durante a tradução.
- Seleção dos componentes:
  - como em um registro: α + deslocamento.
  - sua RA é similar a um OD do tipo registro.
- Criação dinâmica:
  - um registro a cada chamada do subprograma, sendo destruído ao término da rotina.
- Recursão
  - a cada chamada recursiva é criado um novo registro de ativação.

Definição e invocação de subprograma

Subprograma genérico

- Único nome, denominando subprogramas distintos com diferentes assinaturas:
  - ⇒ sobrecarregado,
  - individualização do subprograma é feita pelo tradutor:
    - com base em seus argumentos.
  - traz vantagens sem complicar a implementação
- É uma propriedade central em LP tipo ML (tipos polimórficos) ou Prolog.

Subprograma Genérico
Exemplo em ML

```
fun  size [ ]          = 0
       | size (x::xs) =  1 + size xs;

> val size = fn: ‘a_list  —–> int
size [“a”,”perigo”, “João”, 2,3,4,5]
> 7: int
```

Subprograma Genérico

Exemplo em ADA

```
procedure entra (aluno: in integer;
                 turma: in out Tturma) is
    begin
      ....
    end;
procedure entra(turma: in Tturma;
                tab: in out TlistaTurma) is
     begin
       ...
     end;
```

Subprograma Genérico

Exemplo em Fortran 90

```
INTERFACE ENTRA
   SUBROUTINE ENTRA1 (ALUNO,TURMA)
      INTEGER :: ALUNO
      TTURMA  :: TURMA
            ....
   END SUBROUTINE ENTRA1
   SUBROUTINE ENTRA2 (TURMA,TAB)
      TTURMA :: TURMA
      TLISTATURMA :: TAB
            ...
   END SUBROUTINE ENTRA2
END INTERFACE ENTRA
```

Encapsulamento p/ Subprograma

Definição de subprograma como OD

- Ausente em LP compiladas (C, Pascal, etc.):
  - a definição do subprograma é independente da sua execução,
  - o fonte é compilado, gerando o código executável,
  - durante a execução, a parte estática da definição do subprograma é inacessível e invisível .
- Em LP interpretadas:
  - a definição do subprograma é um objeto de dados executável.

Encapsulamento p/ Subprograma

Definição de subprograma como OD

- Em Lisp e Prolog não há a fase de geração de código e a fase de execução:
  - há facilidades para tratar definição de subprograma como um OD em tempo de execução.
- Exemplo em Lisp

```
$ (defun q2(x) (* x x))
   Q2
$ (mapcar  ‘q2  ‘(1 3 5 7))
   (1 9 25 49)
```

Encapsulamento p/ Subprograma

Definição de subprograma como OD

- Tradução:
  - o tradutor pega o fonte, definido na forma de string de caracteres, e produz um OD run time, representando a definição contida no programa.
- Execução
  - operação que pega o OD run time, cria um registro de ativação e executa o OD run time.
- Em linguagens interpretadas a tradução é considerada uma operação que pode ser invocada em tempo de execução.

Encapsulamento p/ Subprograma

Definição de subprograma como OD

```
(defun ler-fn()
  (let  ( (f nil) (Args nil))
      (print "Entre com a definição de F")
      (if (null (setq f (eval (read))))
          (return-from ler-fn nil))
      (loop  (terpri)
         (print `(Entre com argumentos de ,F))
         (if (null (setq Args (read))) (return nil))
          (print "Resultado =")
          (princ (apply f Args))
      )   ; terpri := nova linha saída padrão. Retorna nil
  ))      ; loop := laço incondicional.
```

Abstração I: Encapsulamento

Definição de tipos

- Definir tipos abstratos de dados requer:

  a) definir um novo tipo ou classe de dados.

  - C, Pascal e Ada permitem definição de tipos
    nome do tipo ⇒ nome da classe de objetos de dados

```
type racional = record
        numerador, denominador: integer;
end;
var A, B, C: racional;
```

  b) mecanismos para definir as operações sobre o novo TD.

  - especialidade de linguagens orientadas a objetos.

Definição de Tipos

Exemplo em C

struct racional {
  int numerador, denominador;}
struct racional A, B, C;

- uso de struct para definir variáveis viola o princípio:
  - tudo que o programador vê é o nome do tipo e a lista de operações para manipulá-lo.

typedef struct {**int** numerador, denominador;}
  racional;
racional A, B, C;

- sintaxe semelhante ao Pascal: não viola a regra de TAD
- typedef é similar a uma macro:
  - novos tipos são criados em C pela declaração struct.

Definição de Tipos

- Uso: gabarito para definir OD na execução:
  - permite separar a definição da estrutura do OD da definição dos pontos nos quais OD desse tipo serão criados na execução do programa.
- Constitui uma forma de encapsular dados e esconder informação:
  - se subprograma cria OD usando o nome do tipo e não acessa diretamente os seus componentes internos ⇒ a definição do tipo pode mudar sem ser necessário alterar o programa.

Definição de Tipos
Exemplo em C

FILE é uma estrutura. Qual é a definição dessa estrutura? Onde é declarada? É necessário conhecê-la para usá-la?

```
FILE *fopen (char *externo, char *modo); // r, w, a, rb, wb, ab
int getc (FILE *fp); // retorna próximo caracter do arquivo fp
int putc (char x, FILE *fp); // grava x em fp; retorna x ou EOF
int fprintf (FILE *fp, “fmt”, arg1, arg2, ...)
int fscanf (FILE *fp, “fmt”,ap1, ap2, ...);
char *fgets (char *linha, int max, FILE *fp); // linha ou NULL
int *fputs (char *linha, FILE *fp); // retorna 0 ou EOF
int fclose (FILE *arquivo);
```

Definição de Tipos
Implementação

- Similar a declarações, *definição de tipo* é útil apenas na tradução para determinar a RA do OD, checagem de tipo e gerência de armazenamento:
  - durante a compilação, definições de tipo são inseridas em uma tabela específica e usadas quando OD deste tipo são declarados.
  - gerado o código objeto, não são necessárias:
    - em LP compiladas os objetos de dados não tem etiqueta de tipo.

Definição de Tipos
Equivalência de Tipos

- Checagem de tipo (estática ou dinâmica)
  - comparação entre os tipos dos argumentos *atuais* e esperados (*formais*) de uma operação.
  - um erro ou coerção ocorre, se os tipos não são os mesmos.
- Questões em equivalência de tipos:
  - quando dois OD são do *mesmo tipo*?
    - questão tratada em tipos de dados.
  - quando dois OD do mesmo tipo são *iguais*?
    - questão semântica relativa ao r-value de um OD.

Definição de Tipos

Equivalência de Tipos

```
program Pxy ;
  type T1= array[1..10] of
    real;
        T2= array[1..10] of
    real;
   var x,z: T1; y: T2;
  procedure Sub(A:T1);
      begin
        ...
      end;
  begin     {ADA,  C++}
    x:=y;    {erro: não Pascal}
    Sub(y);{erro tipo Pascal}
  end ;
```

a) x, y, z e A tem o mesmo tipo?

b) É válido fazer

x:=y? e Sub(y)?

c) Há 2 enfoques básicos:

- equivalência de nomes.
- equivalência estrutural.

d) Equivalência de nomes:

TD são equivalentes se têm o *mesmo* nome.

- usado em ADA, C++ e em passagem de parâmetros em Pascal.

Equivalência de Tipos
Equivalência de nomes

- Desvantagens:
  - todo OD usado em atribuição *precisa* ter um nome de tipo. Não pode haver tipos anônimos.

    var w: **array** [1..10] **of real**;
    - não é argumento válido de subprograma.
  - TD de argumento transmitido em uma cadeia de subprogramas não pode ser novamente definido em cada subprograma:
    - única definição global de tipo deve ser usada.
      - definição de classe em C++, package em ADA e arquivos de inclusão “.h”em C garantem isto.

Equivalência de Tipos
Equivalência Estrutural

- Dois tipos de dados são equivalentes se definem OD com os *mesmos* componentes internos:
  - isto é, ambos tem a mesma RA, em tempo de execução e mesma forma para selecionar componentes.
  - T_1 e T_2 são equivalentes estruturais porque têm a mesma RA.

Equivalência de Tipos

Questões sobre equivalência estrutural

- Registros são equivalentes se componentes:
  - estão na mesma ordem e têm os mesmos tipos?
  - mesma ordem, têm os mesmos tipos e nomes?
  - tendo mesmos tipos e nomes, precisam estar na mesma ordem?
- Arranjos com mesmo nº de componentes de tipos iguais: subscritos precisam ser iguais?
- Literais em dois tipos de enumerações precisam ser os mesmos e estarem na mesma ordem?

Equivalência de Tipos

Questões sobre equivalência estrutural

- Equivalência equivocada:

```
type Tmetro = integer;
     Tlitro  = integer;
var dis: Tmetro;   -  distância em metros
    vol: Tlitro;   -  volume em litros
X:= dis + vol ;    - Equivalência estrutural: OK.
                   - Equivalência de nome: erro.
```

  - O programador declarou tipos diferentes e considerá-los iguais é um erro semântico.

- Compilação cara (freqüência da)

  verificação de equivalência entre OD complexos

Equivalência de Tipos

Projeto de linguagens

- Em LP como Pascal e Ada equivalência de tipos tem um papel central.
- Em LP antigas (Cobol, Fortran, PL/1), não há definição de tipos; usa-se formas de equivalência estrutural.
- C usa equivalência estrutural.
- C++ e Ada usam equivalência de nomes.
- Equivalência é um assunto de pesquisa.

Equivalência de Tipos
Igualdade de OD

- Se dois objetos A e B têm o mesmo tipo, quando são iguais? Isto é A = B?
  - Resposta é não trivial

struct tpilha {**int** topo; **int** dados[100]} x,y;
struct tconjunto {**int** num; **int** dados[100]} a,b;

1) a, b, x e y são estruturalmente equivalentes.
2) as condições sobre as quais se deseja ter a=b e x=y podem ser muito diferentes.

Equivalência de Tipos

Igualdade de OD

- Igualdade de pilha
  - i) x.topo=y.topo // aponta para mesmo índice.
  - ii) x.dados[k]=y.dados[k], para k=0,x.topo-1.
- Igualdade de conjuntos
  - i) a.num = b.num // mesmo # de elementos.
  - ii) a.dados[0], ..., a.dados[a.num-1] é uma permutação (b.dados[0],..,b.dados[a.num-1])
- Não é fácil formalizar igualdade p/ ODE, com tipos criados pelo usuário:
  - cabe ao usuário essa tarefa.

Definição de Tipos

Definição de tipos com parâmetros

- Tipo parametrizado facilita definir tipos de dados similares, p.ex. arrays de ≠ tamanhos

1) exemplo com parâmetros *explícitos* em Ada

```
type mês (dias: integer range 1 .. 31) is
        record
            nomes: string(1 .. 3);
            vendas: array(1 .. dias) of float
        end record;
fev: mês(28); m30: mês(30); m31: mês(31);
```

Número de dias faz parte da declaração do mês.

Definição de Tipos

Definição de tipos com parâmetros

2. Exemplo sem parâmetros *explícitos* em Ada

a) type matriz is array (integer range <>, integer range <>) of float;

- tabela: matriz(1..10,1..20);

b) novalista (1..n) of tipox;

- Cada vez que o subprograma contendo esta declaração for ativado tem-se um arranjo novalista de comprimento n, variável externa ao subprograma.

- Implementação
  - definição funciona como um molde (template). O parâmetro deve ser definido antes de usado.

Abstração I: Encapsulamento
Administração de memória

- Algumas características ou restrições de uma LP são explicadas pela técnica de gerência de memória usada:
  - a) restrições à recursão em Fortran:
    usa gerência estática de armazenagem.
  - b) Pascal: permite gerência baseada em stack.
  - c) Lisp: projeto para usar coletor de lixo.
  - Tendência: considerar as técnicas de gerência de memória dependentes da máquina e não abordá-las em manuais de LP. Programador?

Administração de Memória

Elementos armazenados em run time

- Segmento de código.
- OD e constantes definidos pelo usuários.
- Programas do sistema: rotinas de bibliotecas, rotinas de gerência de armazenamento, interpretadores, tradutores, ...
- Reg. de ativação de subprogramas; dados locais.
- Ponto de retorno de subpgramas e buffers de E/S.
- Variáveis temporárias em avaliação de expressões ou passagem de parâmetros em subprogramas.
- Operações para criar/destruir OD dinâmicos.
- Operações para inserir e destruir componentes.

Gerência de Armazenagem

Controle p/ programador e p/ sistema

- Há LP que deixam a cargo do programador parte da carga de gerência (p. ex. C com malloc/free).
- Há outras LP que não permitem ao programador nenhum controle direto sobre armazenamento.
- A dificuldade em deixar parte do controle ao cargo do programador é dupla:
  - é uma carga grande e às vezes indesejável.
  - pode interferir com o controle feito pelo sistema.
- O controle, como em C, já traz problemas: geração potencial de lixo e referências pendentes.

Gerência de Armazenagem

Controle p/ programador e p/ sistema

- Vantagem do controle pelo programador:
  - sabe quando uma área precisa ser alocada ou pode ser liberada.
- Decidir quem vai controlar não é uma tarefa trivial para os projetistas de LP.
  - Delegar ao programador ⇒ riscos de erros, mas administração e execução mais rápida? Ou impor tipagem forte e delegar a gerência de armazenagem ao sistema ⇒ queda no desempenho da LP?
  - Este é um dos debates fundamentais na comunidade de Engenharia de Software.

Gerência de Armazenagem

Fases

- Alocação Inicial:
  - no início da execução os blocos de alocação podem estar disponíveis ou alocadas.
  - os blocos livres estão disponíveis para alocação dinâmica, quando necessário.
  - todo sistema de armazenagem requer técnicas de gerência para controlar os blocos livres e mecanismos para alocá-los durante a execução, quando necessário.
- Recuperação
- Compactação e reuso

Gerência de Armazenagem

Fases

- Recuperação:
  - requer procedimentos para recuperar blocos que estavam em uso e que se tornaram disponíveis e podem ser liberados para reuso.
  - pode ser muito simples, como atualizar o ponteiro duma pilha ou complexo como um coletor de lixo.

Gerência de Armazenagem

Fases

- Compactação e Reuso
  - O bloco recuperado pode estar pronto para uso ou pode ser necessário compactar a área de armazenagem para obter blocos maiores.
  - Reuso envolve os mesmos mecanismos de alocação inicial.

Gerência de Armazenamento

Técnicas usuais

- Diversas técnicas usuais são utilizadas na implementação de LP. Veremos apenas:
  - alocação estática,
  - baseada em stack,
  - heap com elementos de tamanho fixo,
  - heap com elementos de tamanho variável.
- Para cada uma delas, serão abordadas as fases de alocação inicial, recuperação e compactação e reuso.

Gerência de Armazenamento

Alocação estática

- Técnica mais simples.
- É determinada em tempo de tradução e fica fixa através da execução.
  - Exemplo:
    - segmento de código (usuário e do sistema), buffers de E/S e OD do sistema.
- Não há gerência de armazenagem em tempo de execução:
  - não implementa recuperação e reuso.

Alocação Estática

Implementação usual em Fortran

- Toda alocação de memória é estática.
- Subprogramas são compilados em separado.
- A compilação de cada subprograma cria um segmento de código com registro de ativação, áreas de dados, temporários, etc.
- O carregador aloca espaço para esse segmento e para as rotinas do run time, durante a carga.
- É eficiente e incompatível com recursão.
- Cobol também usa alocação estática e C permite a criação de dados estáticos por razões de eficiência.

Gerência de Armazenamento

Gerência baseada em stack

- É a técnica de alocação dinâmica mais simples para administrar armazenagem dinâmica:
  - no início da execução, um bloco seqüencial é alocado para servir como uma pilha.
  - controle: apenas um ponteiro para o topo da pilha
  - área do topo é livre e pode ser alocada. Para liberar, basta decrementar o ponteiro do tamanho da área liberada.
  - compactação e reuso ocorre automaticamente.
- A ativação de um subprograma implica criar um registro de ativação no topo da pilha.

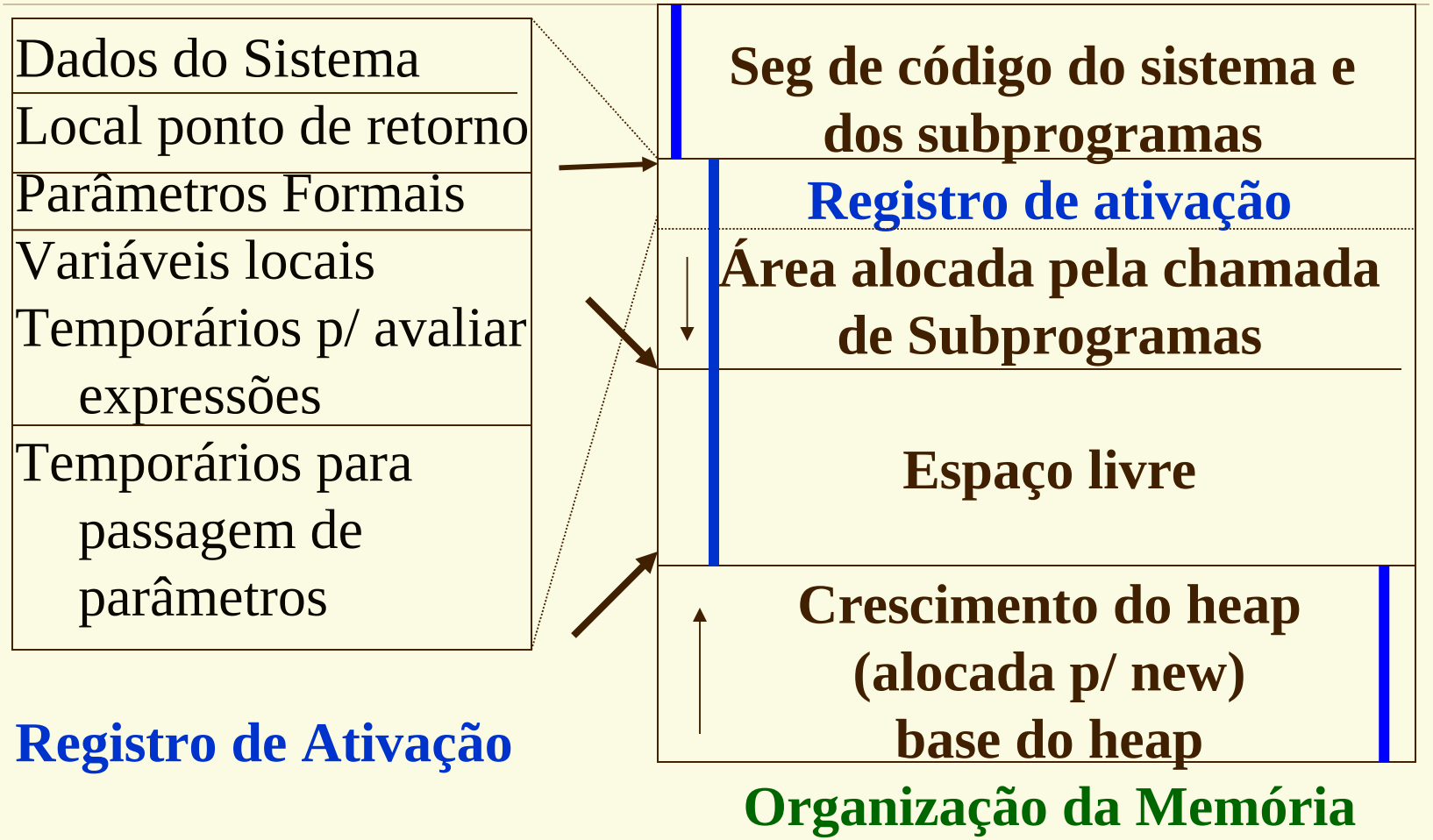
Gerência Baseada em Stack

Exemplo em Pascal

- Em geral usa uma única pilha para ativação de subprogramas e uma área estática contendo programas do sistema e segmentos de código (inclusive dos subprogramas).
- O registro de ativação contém todos os itens de informação associados com a ativação do subprograma.
- O heap é a área para alocação dinâmica via new e dispose.

Gerência Baseada em Pilha

Exemplo em Pascal



Gerência Baseada em Pilha

Exemplo em Lisp

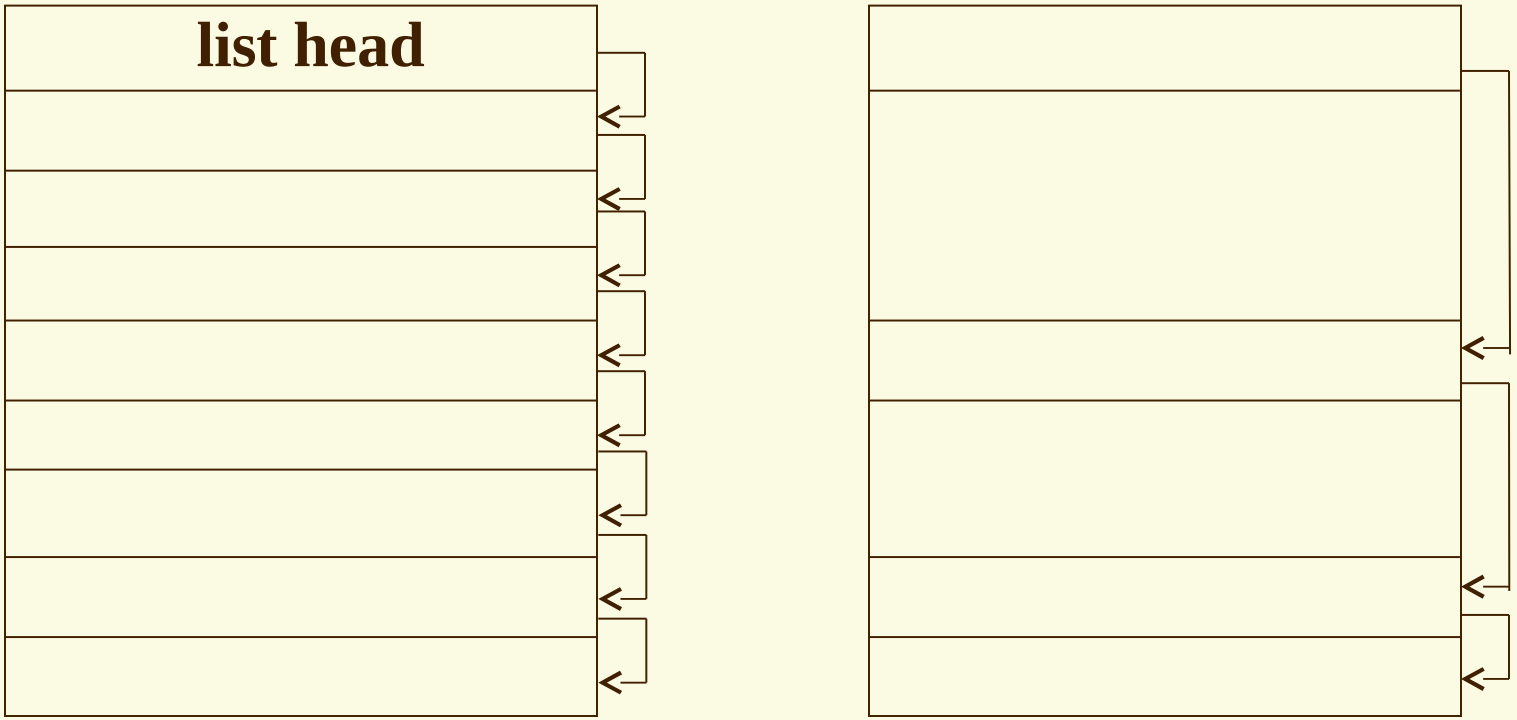
Gerência de Armazenamento

Heap com elementos de tamanho fixo

- Área de memória na qual blocos são alocados e liberados de forma não usual.
  - requerido se a LP permite alocação dinâmica.
  - blocos de tamanho fixo simplificam problemas de alocação, recuperação, compactação e reuso.
    - a área para o heap é contígua e os blocos possuem em geral comprimento de uma ou duas palavras.
    - todos os blocos são encadeados formando uma lista chamada avail ou free-space.
    - alocação: o primeiro elemento de avail é removido da lista, retornando um ponteiro para ele.
    - elementos liberados são encadeados no topo de avail

Heap com OD de tamanho fixo

Estrutura da lista avail



Gerência de Armazenamento

Heap com elementos de tamanho fixo

- Retorno para a lista avail é simples desde que se saiba quais blocos podem ser liberados:
  - problema:
    - determinar quais blocos no heap podem ser recuperados.
- Técnicas:
  - retorno *explícito* pelo programador ou sistema.
    - pode causar referências pendentes e lixo.
  - contador de referências.
    - requer retorno explícito com controle do número de ponteiros para cada bloco para evitar esses problemas.
  - coletor de lixo (garbage collection).
    - permite que lixo seja gerado com o objetivo de evitar referências pendentes.
    - fases: eliminação lógica, marcação e recuperação.

Gerência de Armazenamento

Heap: elementos de tamanho variável

- Assunto para
  SISTEMAS OPERACIONAIS

  - MÉTODO DO MELHOR AJUSTE
  - MÉTODO DO PRIMEIRO AJUSTE
  - MÉTODO DO PIOR AJUSTE

  - REQUER COMPACTAÇÃO PARA EVITAR FRAGMENTAÇÃO DA MEMÓRIA.