

RESUMO

DE

LINGUAGENS DE PROGRAMAÇÃO I

SUMÁRIO

Capítulo I - Introdução.....	3
I.1 O que é uma linguagem de programação.....	3
I.2 Motivos para se estudar os conceitos de linguagens de programação.....	3
I.3 Requisitos desejáveis em uma linguagem de programação.....	3
I.4 Domínios de programação.....	4
I.5 Paradigmas de linguagens de programação.....	4
I.5.1 Linguagens Assertativas.....	5
I.5.1.1 O Paradigma Imperativo.....	5
I.5.1.2 O Paradigma Orientado a Objetos.....	6
I.5.2 Linguagens Declarativas.....	7
I.5.2.1 O Paradigma Lógico.....	7
I.5.2.2 O Paradigma Funcional.....	11
I.6 – LISP.....	12
I.6.1 – Notação.....	12
I.6.2 – Predicado.....	13
I.6.3 – Funções Nativas.....	14
I.6.4 – Operadores Lógicos.....	14
I.6.5 – Definição de Funções.....	15
I.6.6 – Expressões Condicionais.....	16
I.6.7 – Recursividade.....	17
I.7 – HISTÓRICO.....	18
I.8 Sintaxe x Semântica.....	19
I.9 Funcionamento das LP's.....	19
I.9.1 Compilação.....	20
I.9.2 Interpretação.....	21
I.9.3 Mecanismos híbridos.....	21
I.10 Lista de Exercícios.....	22
Capítulo II - A Memória.....	24
II.1 – Alocação de memória.....	24
II.2 – Ponteiros.....	25
II.3 Passagem de Parâmetros.....	27
II.3.1 Passagem por valor.....	27
II.3.2 Passagem por referência.....	28
II.4 Funcionamento da Memória.....	30
II.5 Exercício.....	30
Capítulo III - Valores e Tipos.....	34
III.1 – Valores.....	34
III.2 – Tipos.....	34
III.2.1 Tipos Primitivos.....	34
III.2.1.1 Inteiro.....	35
III.2.1.2 Ponto-flutuante.....	35
III.2.1.3 Decimais.....	36
III.2.1.4 Booleanos.....	36
III.2.1.5 Caractere.....	36
III.2.2 Tipos compostos.....	36
III.2.2.1 Mapeamentos.....	36
III.2.2.2 Vetor.....	37
III.2.2.3 String.....	40
III.2.2.4 Tipo Ponteiro.....	41
III.2.2.5 Registro.....	44
III.2.2.6 União disjunta.....	45

III.2.2.7 Tipos Enumerados.....	47
III.2.2.8 Tipos Sub-faixa.....	49
III.2.2.9 Tipo Conjunto.....	50
III.3 Verificação de Tipos.....	51
III.4 Lista de Exercícios.....	54
Capítulo IV - Expressões e Atribuição.....	56
IV.1 Expressões aritméticas.....	56
IV.1.1 Ordem de avaliação de operadores.....	56
IV.1.2 Ordem de avaliação dos operandos.....	57
IV.2 Expressões Relacionais.....	60
IV.3 Expressões Booleanas.....	60
IV.4 Avaliação Curto-Circuito.....	60
IV.5 Atribuição.....	60
IV.6 Lista de Exercícios.....	62
Capítulo V – VARIÁVEIS.....	64
V.1 Nomes.....	64
V.2 Endereço.....	65
V.3 Tipo.....	65
V.4 Valor.....	65
V.5 Tempo de vida.....	66
V.6 Escopo.....	68
V.7 Constantes Nomeadas.....	71
V.8 Inicialização de Variáveis.....	71
V.9 Variáveis Transientes e Variáveis Persistentes.....	71
V.10 Lista de Exercícios.....	71
Capítulo VI - Abstração de Controle.....	75
VI.1 – Instruções de seleção.....	75
VI.1.1 – Seletores Bidirecionais:.....	75
VI.1.2 – Seletores Múltiplos.....	76
VI.2 – Instruções Iterativas.....	76
VI.2.1 – Laços controlados por contador:.....	77
VI.2.2 – Laços controlados logicamente:.....	78
VI.2.3 – Mecanismos de controle de laços localizados pelo usuário.....	79
VI.3 – Desvio incondicional.....	79
VI.4 – Comandos protegidos:.....	79
VI.5 Lista de Exercícios.....	80
Capítulo VII - Abstração de Processos.....	83
VII.1 – Introdução.....	83
VII.2 – Definições.....	83
VII.3 – Parâmetros.....	83
VII.3.1 – Tipos de Parâmetros.....	84
VII.3.2 – Métodos de passagem de parâmetros.....	84
VII.3.3 Verificação de tipos dos parâmetros.....	86
VII.4 – Módulo.....	86
VII.5 Abstração de Dados – Tipos Abstratos de Dados.....	86
VII.6 Subprogramas Sobrecarregados (<i>OVERLOADING</i>).....	88
VII.7 – Subprogramas Polimórficos – Polimorfismo paramétrico.....	89
VII.8 Classes.....	89
VII.9 Lista de Exercícios.....	91

CAPÍTULO I - INTRODUÇÃO

I.1 O que é uma linguagem de programação

Notação para descrever ordens que o computador deverá executar.

I.2 Motivos para se estudar os conceitos de linguagens de programação

- **Melhora a compreensão das linguagens já conhecidas** ⇒ Assim, ao conhecer várias linguagens, o número de limitações impostas ao construir um software é reduzido, pois o programador torna-se capaz de aprender novas construções de linguagem.
- **Facilita o aprendizado de novas linguagens** ⇒ Ao adquirir uma compreensão dos conceitos fundamentais das linguagens, será mais fácil compreender como esses estão incorporados ao projeto da linguagem aprendida. Por exemplo, programadores que entendem o conceito de abstração de dados aprenderão construir, melhor e mais rápido, tipos abstratos de dados em Java.
- **Facilita o projeto de uma linguagem** ⇒ Um exame crítico das LP's ajudará no projeto de sistemas complexos, ajudando os usuários a examinar e avaliar esses produtos.
- **Facilita a escolha de uma linguagem para um projeto** ⇒ Pois cada linguagem tem sua característica própria, bem como aplicações próprias.

I.3 Requisitos desejáveis em uma linguagem de programação

- Legibilidade ⇒ Um software deve ser facilmente lido e entendido, pois a manutenção é uma parte importante. A linguagem não deve permitir implementar código ilegível. Para isto, a linguagem deve ter:
 - ✓ **Simplicidade** (por isso, ela **NÃO** deve ter muitos comandos, um único símbolo com vários significados, sobrecarga de operadores, para uma função: `c = c + 1;` `c++;` `c += 1;` `++c`)
 - ✓ **Ortogonalidade**, ou seja, um conjunto pequeno de instruções primitivas pode ser combinado para construir as estruturas de controle e de dados da linguagem. Além disso, toda combinação de primitivas deve ser legal e significativa. A falta de ortogonalidade acarreta exceções às regras da LP.
 - ✓ **Instruções claras de controle**, para isso, deve ser evitado o uso de GOTO.
 - ✓ **Tipos de Dados e Estruturas adequados** (Ex.: ter um tipo booleano e não um tipo inteiro como sinalizador, ter um tipo registro e não usar um conjunto de vetores)
 - ✓ **Boa sintaxe**, (contra-exemplos: restringir identificadores a tamanhos muito pequenos, não ter palavras especiais para identificar início e/ou final de blocos, permitir letra maiúscula e minúscula, permitir que palavras especiais possam ser usadas como nomes de variáveis, ter instruções com nomes não significativos ou duas instruções com nome idênticos mas com significados diferentes).

- Facilidade de escrita (Redigibilidade) ⇒ Facilidade com que a linguagem pode ser usada para criar programas. Portanto, a linguagem não deve ter um número grande de construções primitivas e um conjunto consistente de regras para combiná-las.
- Tratamento de exceções ⇒ Verificar erros em tempo de execução e pôr em prática medidas corretivas
- Eficiência
- Reuso de código ⇒ Subprogramas, modularização, estrutura de dados
- Flexível ⇒ Facilidade de modificar o programa a partir de novos requisitos
- Fácil de aprender

□ **Questionamento:** Uma LP que abrangesse todos os tipos de dados possíveis, todos os operadores possíveis, possuísse qualquer tipo de comando implementado seria considerada uma boa linguagem de programação? Sim ou Não? Por quê?

I.4 Domínios de programação

Uma LP pode ser desenvolvida, criada para determinados fins, determinados domínios. Podemos citar alguns destes domínios:

- **Aplicações comerciais – COBOL (1960)** ⇒ Possuem facilidade para produzir relatórios elaborados
- **Aplicações científicas – FORTRAN (1957)** ⇒ Possuem estrutura de dados simples, mas exigem um grande número de computações aritméticas com números reais. As estruturas de dados mais comuns são matrizes e as de controle são os laços de contagem e seleções.
- **Programação de novos sistemas – PL/S; C** ⇒ Devem ser eficientes na execução, ou seja, uma execução rápida. Além disso, devem ter recursos de baixo nível que permitam ao software fazer interface com dispositivos externos.
- **Linguagens de scripting – AWK; KSH** ⇒ São usadas colocando-se uma lista de comandos, chamados de *script*, em um arquivo para serem executados.
- **Linguagens de propósitos especiais – RPG** ⇒ produzir relatórios comerciais

I.5 Paradigmas de linguagens de programação

Um paradigma é um modelo, padrão ou estilo de programação suportado por determinado grupo de linguagens.

A maioria dos livros da área de LP categoriza 4 paradigmas de programação:

- O paradigma imperativo
- O paradigma funcional
- O paradigma lógico
- O paradigma orientado a objetos

I.5.1 Linguagens Assertativas

São linguagens que se baseiam em expressões as quais modificam valores de entidades (dados ou objetos). Dividem-se em dois grupos:

- **Linguagens imperativas ou orientadas a dados**, em que a construção de programas é dirigida pelas transformações que ocorrem nos dados.
- **Linguagens orientadas a objetos**, em que a construção de programas é dirigida pelas mudanças de estados de objetos.

I.5.1.1 O Paradigma Imperativo

A maioria das LP's dos últimos foi projetada em função da arquitetura de computador prevalecente, chamada de arquitetura Von Neumann (pronuncia-se fon Nóiman) em homenagem a um de seus criadores, John von Neumann.

Nesta arquitetura, tanto os dados como os programas são armazenados na mesma memória, vide figura 1. A CPU, que executa realmente as instruções, é separada da memória. Os dados e instruções são levados à CPU e, em seguida, o resultado é transferido para a memória.

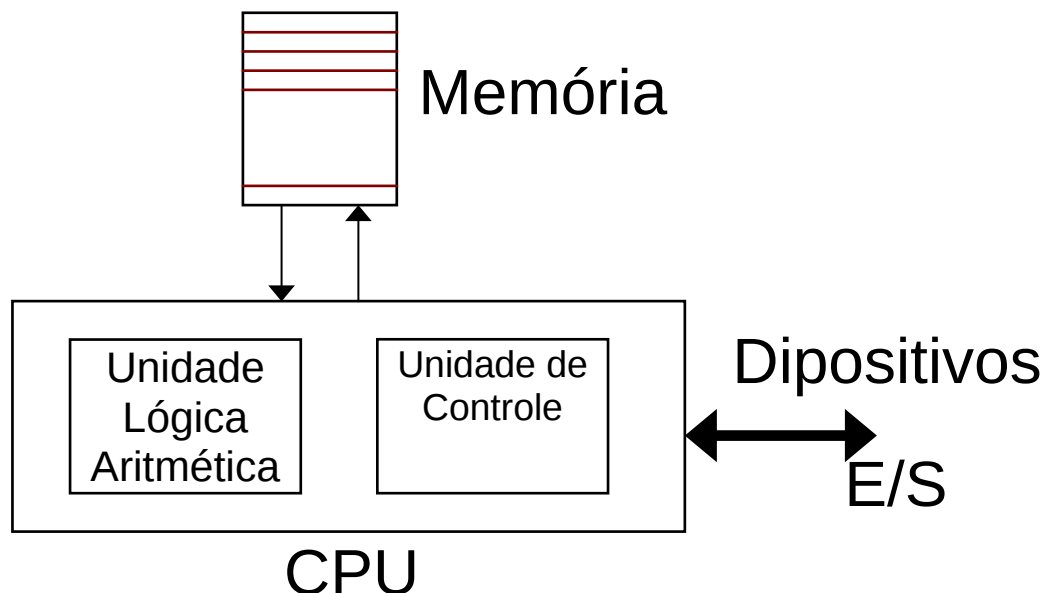


Figura 1: Arquitetura da Von Neumann

Assim as linguagens imperativas modelam esta arquitetura através de variáveis, que modelam as células de memória, e comando de atribuição, responsáveis pela transmissão entre memória e CPU. Desta forma os programas feitos nestas linguagens são muito eficientes, pois extraem ao máximo as funcionalidades da máquina.

Os programas são centrados no conceito de comandos e atualizações de variáveis

Paradigma também denominado de **procedural**, pois a programação é baseada em blocos de comando e de sub-rotinas ou procedimentos, que fazem a estruturação.

A velocidade de conexão entre a memória do computador e seu processador determina a velocidade de um computador. Observe que as instruções geralmente são executadas mais rapidamente do que podem ser transferidas. Assim, o fator principal que limita a velocidade de um computador é conhecido como **gargalo de Von Neumann**.

Exemplos de LP's imperativas: FORTRAN, PASCAL, C, etc..

Vantagens:

- Eficiência (arquiteturas atuais são Von Neumann)

- Paradigma mais do que estabelecido

Desvantagens:

- Díficil legibilidade, pois os programas são escritos no formato top-down
- Introdução de erros durante a manutenção
- Descrições demasiadamente operacionais focalizam o **como** um processo deve ser realizado na máquina.

I.5.1.2 O Paradigma Orientado a Objetos

Este paradigma é uma subclassificação do Imperativo, sendo mais usados com este, embora existam, por exemplo, versões de Lisp OO como CLOS. A diferença entre os dois paradigmas está na metodologia de concepção e modelagem do sistema.

De maneira simplista podemos dizer que LP's OO permitem abstração de dados, com a criação de um tipo (classe), que, além de conter variáveis (atributos), também possuirá, na sua declaração, os procedimentos ou funções (métodos) para manusear este tipo. Uma variável deste tipo poderá ser criada, e esta é chamada então de objeto. Qualquer atualização ou alteração da variável se dará mediante os métodos declarados no tipo.

A programação é desenvolvida através de trocas de mensagens entre os objetos, que são ligados a uma classe. As classes, por sua vez, se interligam através de heranças.

Há ênfase em reutilização de código, principalmente através de **herança** e **polimorfismo**.

Exemplo em JAVA:

```
class Data {
    private int dia;
    private int mes;
    private int ano;

    public void Alterar_data ( int d, int m, int a) {
        dia = d;
        mes = m;
        ano = a;
    }
}

public class Principal {
    public static void main (String[] args ) {
        Data Data_Nascimento = new Data();
        Data_Nascimento.Alterar_data (1, 1, 1980);
    }
}
```

Vantagens:

- Eficiência (arquiteturas atuais são Von Neumann)
- Classes estimulam projeto centrado em dados: modularidade, reusabilidade e extensibilidade, facilitando o reuso de código
- Ciclo de vida mais longo para os sistemas
- Desenvolvimento mais rápido de sistemas

- Possibilidade de se construir sistema muito mais complexos (incorporação de funções prontas)
- Menor custo para desenvolvimento e manutenção de sistemas

Desvantagens:

- Maior esforço na modelagem de um sistema OO do que estruturado (porém menor esforço de codificação)
- Dependência de funcionalidades já implementadas em superclasses no caso da herança, implementações espalhadas em classes diferentes.

I.5.2 Linguagens Declarativas

São linguagens que se baseiam em *expressões* que verificam ou induzem a que ocorram relações entre declarações. Dividem-se em dois grupos:

- **Funcionais**, em que essas relações são caracterizadas por mapeamentos entre estruturas simbólicas.
- **Lógicas**, em que essas relações são caracterizadas como expressões da lógica matemática.

I.5.2.1 O Paradigma Lógico

No paradigma lógico, os programas são escrito com base nas relações entre entrada e saída. Ou seja, a programação consiste em proposições (instrução lógica que pode ou não ser verdadeira) e conectores lógicos (**e**, **ou**, **negação**, **implica**, **equivale**). Por exemplo: **HOMEM (Carlos)**. Por isso, linguagens deste paradigma utilizam lógica simbólica e um processo de inferência lógica.

A linguagem mais importante que trabalha dessa forma é o **PROLOG**.

Um programa, em Prolog, é constituído de uma coleção de unidades lógicas chamadas predicados. Cada predicado é uma coleção de cláusulas, que por sua vez são fatos ou regras (implicações lógicas – se...então).

Todas as cláusulas necessárias para resolver um determinado problema são inseridas em um arquivo de banco de proposições do Prolog, que é um arquivo texto. Após serem compiladas, elas estão prontas para serem usadas.

O programa roda baseado em perguntas realizadas pelo usuário e que são respondidas usando-se a base de fatos e regras para produzir uma sequencia de deduções lógicas até chegar a solução desejada.

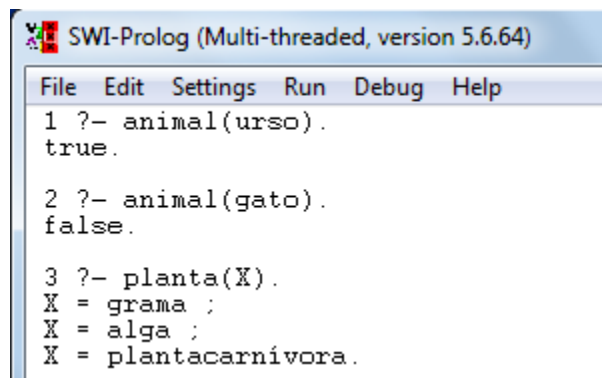
Observe na figura 2 alguns fatos digitados no Prolog:



```
cadeia_alimentar.pl [modified]
File Edit Browse Compile Prolog Pce Help
animal(urso).
animal(peixe).
animal(raposa).
animal(veado).
animal(minhoca).
animal(lince).
animal(coelho).
animal(guaxinim).
animal(mosca).
animal(peixinho).
planta(grama).
planta(alga).
planta(plantacarnívora).
```

Figura 2: Primeiros fatos digitados no Prolog

Com esses fatos salvos no banco de proposições do Prolog e com programa compilado, é possível já fazer algumas perguntas, como pode ser visto na figura 3:



```
SWI-Prolog (Multi-threaded, version 5.6.64)
File Edit Settings Run Debug Help
1 ?- animal(urso).
true.
2 ?- animal(gato).
false.
3 ?- planta(X).
X = grama ;
X = alga ;
X = plantacarnívora.
```

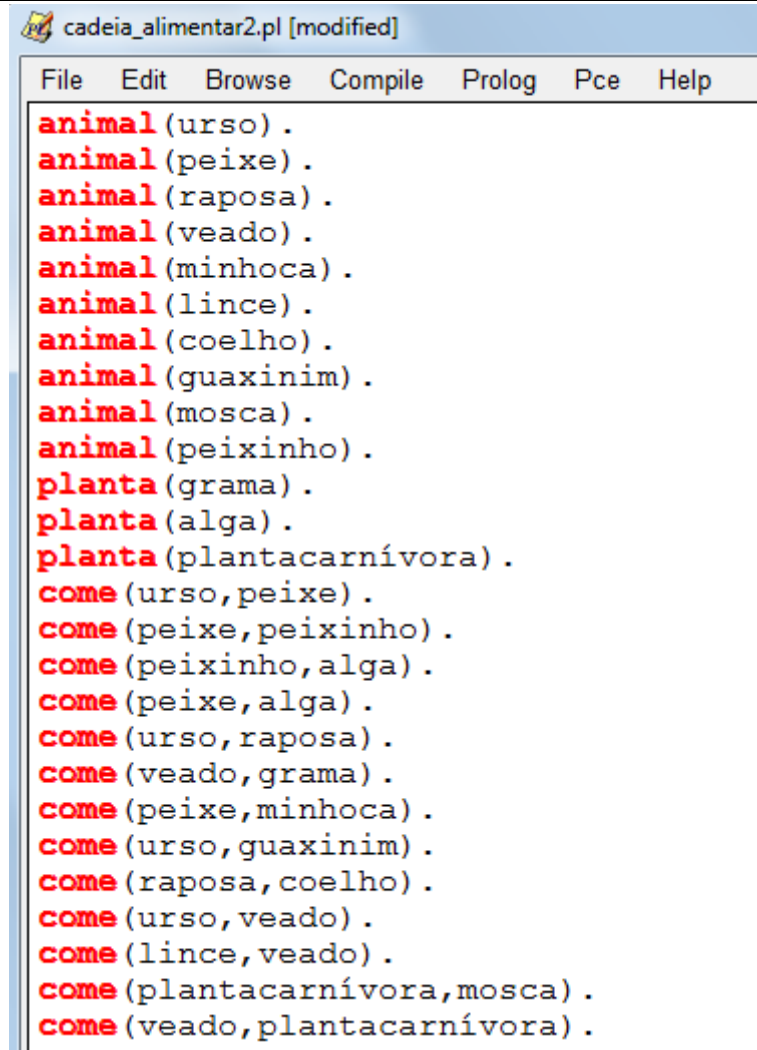
Figura 3: Perguntas feitas ao Prolog

A primeira pergunta é se **urso é um animal**. O Prolog verifica se ele tem essa informação em seu banco de proposições. Como ele tem, a resposta é *true* (verdadeiro).

A segunda pergunta é se **gato é um animal**. Como não há essa informação em seu banco de dados, a resposta é *false* (falso).

A terceira pergunta é **quais os elementos do banco de proposições que são plantas**. Então, ele procura todas as proposições do tipo **planta(...)** e retorna-as.

É possível melhorar este banco de proposições, como vista na figura 4:



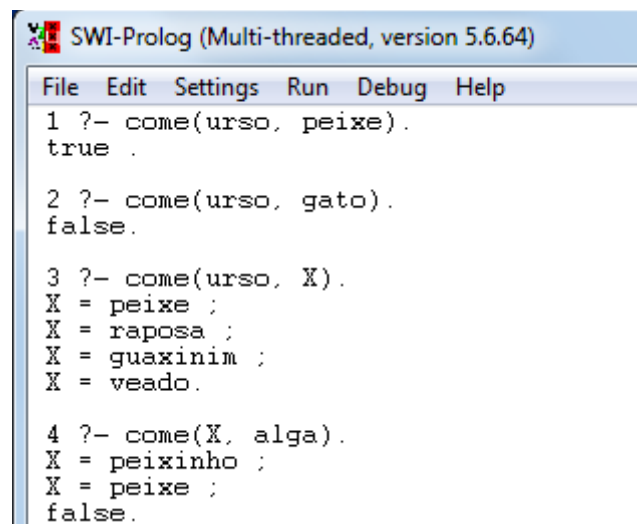
```

animal(urso).
animal(peixe).
animal(raposa).
animal(veado).
animal(minhoca).
animal(lince).
animal(coelho).
animal(guaxinim).
animal(mosca).
animal(peixinho).
planta(grama).
planta(alga).
planta(plantacarnívora).
come(urso,peixe).
come(peixe,peixinho).
come(peixinho,alga).
come(peixe,alga).
come(urso,raposa).
come(veado,grama).
come(peixe,minhoca).
come(urso,guaxinim).
come(raposa,coelho).
come(urso,veado).
come(lince,veado).
come(plantacarnívora,mosca).
come(veado,plantacarnívora).

```

Figura 4: Acréscimo ao Banco de Proposições

Neste momento é possível fazer outros tipos de perguntas, como visto na figura 5.



```

1 ?- come(urso, peixe).
true.

2 ?- come(urso, gato).
false.

3 ?- come(urso, X).
X = peixe ;
X = raposa ;
X = guaxinim ;
X = veado.

4 ?- come(X, alga).
X = peixinho ;
X = peixe ;
false.

```

Figura 5: Novas perguntas

A pergunta 3 terá como respostas **tudo o que o urso come** e a pergunta 4 terá como resposta **todos os animais que comem alga**.

É possível também incluir regras, ou seja, implicações lógicas, junto com os fatos. Por exemplo:

- ✓ “**Carnívoro**”, se X é carnívoro, então X come Y e Y é animal.
- ✓ “**Herbívoro**”, se X é herbívoro, então X come Y e Y é planta e X não é carnívoro.
- ✓ “**X pertence a cadeia alimentar de Y**”, se Y come X ou então se Z pertence a cadeia alimentar de Y e Z come X.

Essas regras podem ser vistas na figura 6.

```
carnívoro(X):-come(X,Y), animal(Y).  
herbívoro(X):-come(X,Y),planta(Y),\+carnívoro(X).  
pertence_a_cadeia(X,Y):-come(Y,X).  
pertence_a_cadeia(X,Y):-come(Z,X),pertence_a_cadeia(Z,Y).
```

Figura 6: Introdução de Regras

Podemos fazer as seguintes perguntas:

1 – Quem pertence a cadeia alimentar do urso?

1 ?- **pertence_a_cadeia**(X,urso).

X = peixe ;

X = raposa ;

X = guaxinim ;

X = veado ;

X = peixinho ;

X = alga ;

X = alga ;

X = grama ;

X = minhoca ;

X = coelho ;

X = mosca ;

X = plantacarnívora ;

false.

2 – Quem pertence a cadeia alimentar do urso e também come planta?

2 ?- **pertence_a_cadeia**(X,urso),**herbívoro**(X).

X = veado ;

X = veado ;

X = peixinho ;

false.

3 – A minhoca pertence a cadeia alimentar de quem?

pertence_a_cadeia(minhoca,X).

X = peixe ;

X = urso ;

false.

E assim a programação vai acontecendo.

É possível encontrar programas deste tipo de paradigma em sistemas de gerenciamento de banco de dados relacionais, sistemas especialistas, IA, prototipagem em geral, dentre outras.

Vantagens:

- Em princípio, todas do paradigma funcional
- Permite concepção da aplicação em um alto nível de abstração (através de associações entre E/S)
- Programas mais lógicos, gerando menos erro e manutenção.
- Permite processamento paralelo
- Tempo de prototipação mínimo

Desvantagens:

- Em princípio, todos do paradigma funcional.
- Linguagens usualmente não possuem tipos nem são de alta ordem.

Observação: Tendência Atual é a combinação de paradigmas

Atualmente as linguagens de programação modernas incorporam um ou mais paradigmas. Por exemplo, Java é uma linguagem imperativa, orientada a objetos e distribuída. Pascal é uma linguagem imperativa. ML é uma linguagem predominantemente funcional, porém possui uma extensão imperativa.

I.5.2.2 O Paradigma Funcional

No paradigma funcional, programas são funções que descrevem uma relação explícita e precisa entre entrada e saída, ou seja, há uma comunicação direta com o programador, sem a necessidade de criar programas formais ou variáveis para realizar operações simples.

O objetivo desse tipo de linguagem é imitar ao máximo as funções matemáticas.

Sua programação é desenvolvida combinando funções simples para formar funções complexas, não havendo conceito de estado (variáveis) nem comandos como atribuição (estilo declarativo). Também não há comandos de iteração, pois não permite variáveis para o controle. Esta será realizada por meio de recursividade.

Linguagens deste paradigma são usadas principalmente em aplicações de IA, além desta área utilizou-se LISP para criar o editor de texto EMACS.

Exemplos: LISP, inicialmente puramente funcional depois adquiriu alguns conceitos imperativos. Scheme, ML, Haskell (puramente funcional)

Vantagens:

- Não precisa se preocupar com variáveis \Rightarrow queda de eficiência
- Programação mais simples
- Execução concorrente é mais simples

Desvantagens:

- “O mundo não é funcional!”
- Implementações ineficientes
- Mecanismos primitivos de E/S e formatação

I.6 – LISP

LISP é uma linguagem bastante dinâmica e fácil de adaptar-se às mudanças de paradigmas de programação que podem ocorrer.

Seus programas são constituídos de pequenos módulos, os quais cumprem objetivos bem simples. Um programa completo é constituído da combinação de seus módulos. LISP é uma linguagem flexível, pois permite que funções definidas pelo usuário não sejam distintas daqueles inerentes à linguagem, além de não haver tipificação de dados.

Essa linguagem é interpretada, o que a tornava muito mais lenta que as outras linguagens. Mas, com o aparecimento de compiladores eficazes e da melhoria do suporte dos processadores, tornou-se mais eficiente.

Ela permite ao programador incorporar outros estilos de programação. Assim, ela adapta-se ao problema que está sendo resolvido.

LISP é uma linguagem interativa, pois cada parte de um programa pode ser compilada, corrigida e testada, de forma independente das demais.

I.6.1 – Notação

Sua notação é prefixa, ou seja, para construir expressões são utilizados primeiro operadores e, após, operandos. Então, o primeiro elemento é considerado um operador e os demais, operandos.

Ex.

NOTAÇÃO INFIXA: $2 + 3 * 5$

NOTAÇÃO PREFIXA: $(+ 2 (* 3 5))$

Vantagens da Notação Prefixa:

a. É possível usar um operador para mais de dois argumentos:

> $(+ 1 2 3)$

6

> $(+ 1 2 3 4 5 6 7 8 9 10)$

55

b. Não existe precedência entre operadores. A precedência é feita através de parênteses:

> $(+ 1 (* 2 3))$

7

> $(* (+ 1 2) 3)$

9

c. Os operadores definidos pelo programador são usados da mesma maneira que os operadores da linguagem (operador primeiro, operandos depois).

Desvantagem:

Escrita de combinações complexas.

Ex.: $(- (+ 1 (* 2 3)) (* (/ 4 5) 6))$

Para facilitar a leitura, é possível indentar uma expressão:

```
(- (+ 1
    (* 2 3))
  (* (/ 4 5)
    6))
```

Podemos fazer as seguintes observações:

1) Divisão de argumentos inteiros retorna inteiro ou fração:

$> (/ 21 7)$	$> (/ 21 7.0)$	$> (/ 9 2)$	$> (/ 9.0 2)$
> 3	> 3.0	$> 9/2$	> 4.5

2) Divisão com um argumento retorna o inverso do argumento:

$> (/ 2.0)$	$> (/ 2)$
> 0.5	$> 1/2$

Exercícios:

1– Converta de notação infixa para prefixa:

- a) $1 + 2 - 3$
- b) $1 - 2 * 3$
- c) $1 * 2 - 3$
- d) $1 * 2 * 3$
- e) $(1 - 2) * 3$
- f) $(1 - 2) + 3$
- g) $1 - (2 + 3)$
- h) $2 * 2 + 3 * 3 * 3$

2 – Calcule as expressões abaixo e depois converta de notação prefixa para infixa:

- a) $(* (/ 1 2) 3)$
- b) $(* 1 (- 2 3))$
- c) $(/ (+ 1 2) 3)$
- d) $(/ (/ 1 2) 3)$
- e) $(/ 1 (/ 2 3))$
- f) $(- (- 1 2) 3)$
- g) $(- 1 2 3)$
- h) $(- 1)$

I.6.2 – Predicado

Predicado é definido como uma função usada como teste. O valor do teste é interpretado como VERDADEIRO ou FALSO. Isso não significa que seja um tipo lógico ou booleano, mas uma expressão condicional considera NIL como FALSO e qualquer outro valor como VERDADEIRO.

Ex.:

```
> (> 4 3)
> t
> (< 4 3)
> NIL
```

I.6.3 – Funções Nativas

Existem funções matemáticas nativas do LISP que facilitam a vida de quem utiliza essa linguagem:

- *(max 2 19 7)* – retorna o maior valor da lista
- *(min 2 19 7)* – retorna o menor valor da lista
- *(expt 2 4)* – 2 elevado a 4
- *(sqrt 9)* – raiz quadrada de 9
- *(mod 4 2)* – retorna o resto inteiro da divisão de 4 por 2
- *(truncate 13.6)* – retorna o valor truncado e o resto
- *(round 13.6)* – retorna o valor arredondado e o resto
- *(abs -2)* – retorna o valor absoluto de -2
- *(random 5)* – retorna um número randômico entre 0 e 5
- **ZEROP** – retorna T se o argumento for 0.

Exemplo: *(ZEROP (- 3 2 1))*

> T

(ZEROP 1)

> NIL

- **EQUAL** – retorna T se existirem dois argumentos e eles forem iguais.

Exemplos: *(EQUAL (+ 2 3) (- 6 1))* T

Note que **EQUAL** avalia resultado de expressões booleanas também.

- **EVENP** – retorna T se o argumento for par.

Exemplo: *(EVENP * 3 17))*

> T

- **ODDP** – retorna T se o argumento for ímpar.

I.6.4 – Operadores Lógicos

AND ⇒ Avalia os argumentos da esquerda para a direita até que um deles seja falso. e nenhum for falso, devolve o valor do último argumento.

OR ⇒ Avalia os argumentos da esquerda para a direita até que um deles seja diferente de falso, devolvendo este valor. Se todos forem falsos, devolve falso.

NOT ⇒ Avalia verdadeiro se o argumento for falso e vice-versa.

OBS.: Lembre-se que qualquer valor diferente de NIL é considerado verdadeiro.

Exercício: Determine o valor das seguintes expressões:

a) *(and (or (> 2 3) (not (= 2 3))) (< 2 3))*

b) (not (or (= 1 2) (= 2 3)))

c) (or (< 1 2) (= 1 2) (> 1 2))

d) (and 1 2 3)

e) (or 1 2 3)

f) (and nil 2 3)

g) (or nil nil 3)

I.6.5 – Definição de Funções

Vamos definir uma função para elevar um número ao quadrado:

SINTAXE:

```
> ( defun nome (par1 par2 ... parn)
    corpo)
```

Onde:

DEFUN ⇒ Informa ao avaliador que estamos definindo uma função

NOME ⇒ Nome da função

PAR ⇒ São os parâmetros passados

CORPO ⇒ Combinação de expressões que determina o valor da função para os parâmetros passados.

```
> (defun quadrado (n)
    (* n n)
  )
quadrado
```

O avaliador acrescenta a função ao conjunto de funções da linguagem, associando ao nome que lhe foi dado e retorna como valor o nome da função.

```
> quadrado 5
25
> quadrado 9
81
> quadrado (+ 1 2)
9
```


No exemplo anterior, o interpretador avalia primeiro o operando $(+ 1 2) = 3$, que é usado pela função, como parâmetro real.

É possível também usar a função mais de uma vez:

```
> (quadrado (quadrado (+ 1 2)))  
(quadrado (quadrado (3)))  
(quadrado (* 3 3))  
(quadrado 9)  
(* 9 9)  
81
```

Etapas de avaliação de
operandos e do corpo da função

Ao definir uma função associamos um procedimento a um nome, precisando de memória para guardar o procedimento em si e a associação. Esta memória do LISP é chamada de AMBIENTE.

O ambiente só existe enquanto se trabalha com a linguagem. Ao terminar, perde-se todo o ambiente. O que fazer, então, para não perder o trabalho?

Escrever as funções em um ficheiro e ir passando-as para o LISP.

Observe que quase não há limitações para a escrita de nomes. Assim, a diferença entre nomes e combinações está na existência de parênteses e espaços em branco.

```
> (defun x+y*z (x y z)  
  (+ (* y z) x))  
x+y*z
```

```
> (x+y*z 1 2 3)  
7
```

O primeiro elemento da combinação o avaliador LISP usa como definição da função.

1.6.6 – Expressões Condicionais

São expressões cujo valor depende de um ou mais testes realizados previamente, onde é possível escolher caminhos diferentes para obtenção do resultado.

No caso mais simples temos 2 alternativas.

Seleção Simples (IF)

SINTAXE:

```
(IF condição  
  consequência  
  alternativa)
```

IF é avaliado determinando o valor da condição. Se for **T**, é avaliado a **consequência**. Se for **NIL**, é avaliada a **alternativa**.

EX.:

```
(IF (> 4 3)  
  5  
  6)
```

EXERCÍCIO:

- 1) Defina uma função soma_grandes que recebe 3 números como argumentos e determina a soma dos dois maiores
- 2) Escreva uma função que calcula o fatorial de um número.

Seleção Múltipla (COND)

É uma espécie de CASE do PASCAL Este comando simplifica o caso de muitos IF(s).

SINTAXE:

```
(COND (cond1  expr1)
      (cond2  expr2)
      ...
      (condn  expren))
```

COND testa cada uma das condições. A primeira verdadeira que encontrar, devolve o valor da expressão correspondente, terminando a avaliação.

EX.:

```
(COND ((> 4 3) 5)
      (T      6))
```

```
2) (defun teste (x y z w)
     (cond ((> x y) z)
           ((< (+ x w) (* y z)) x)
           ((= w z) (+ x y))
           (T 777)))
```

Se esta função for definida com IF, teremos:

```
3) (defun teste (x y z w)
     (if (> x y)
         z
         (if (< (+ x w) (* y z))
             x
             (if (= w z)
                 (+ x y)
                 777)))))
```

I.6.7 – Recursividade

Em funções recursivas deve haver:

- a) Um passo básico com resultado reconhecido de imediato
- b) Um passo recursivo em que se tenta resolver um sub-problema inicial

O objetivo é resolver subproblemas cada vez mais simples até atingir o mais simples de todos, ou seja, aquele que tem resultado imediato. Podem ocorrer ERROS, como por exemplo:

- a) Não detectar o caso mais simples
- b) Não diminuir a complexidade do problema
- c) A recursão pode não parar

Exercício:

1) Faça funções que calculem:

a) $S = 1 + 2 + 3 + \dots + n$

b) $S = \frac{1}{1} + \frac{3}{2} + \frac{5}{3} + \frac{7}{4} + \dots$

c) $\Pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \dots$

d) $S = \frac{100}{0!} + \frac{99}{1!} + \frac{98}{2!} + \frac{97}{3!} + \dots$

e) $e^x = x^0 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots$

I.7 – HISTÓRICO

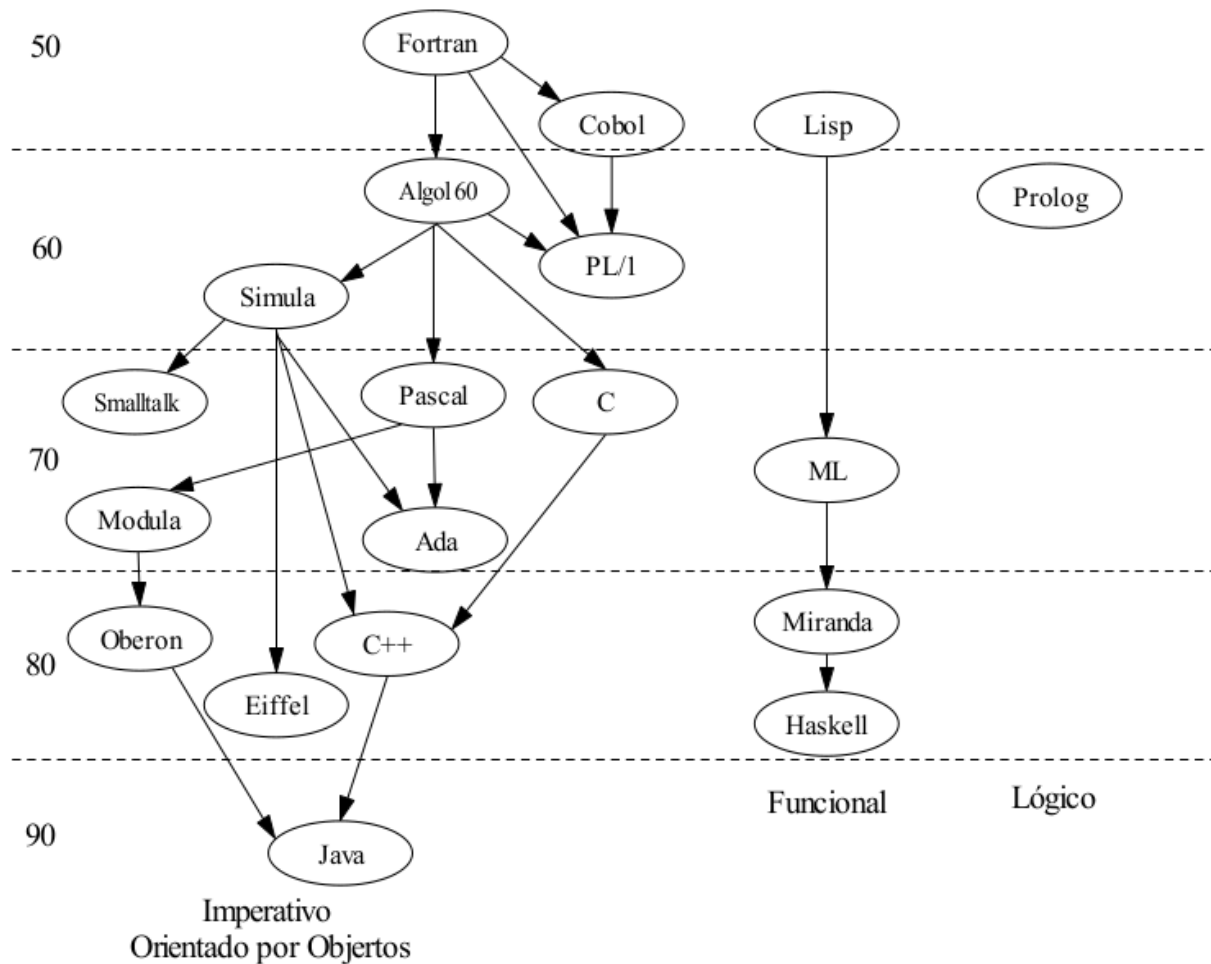


Figura 7: Histórico das Linguagens de Programação

Breve comentário sobre as linguagens da figura 7:

- Fortran é a precursora de todas as linguagens imperativas modernas. Linguagem oficial dos antigos mainframes da IBM. Ainda muito usada em supercomputadores, como o CRAY, por físicos e engenheiros.

- Algol apresentou o primeiro esboço interessante de uma linguagem de programação moderna, incluindo conceitos como estruturas de blocos e procedimentos. Não saiu dos limites da academia.
- Cobol foi criada para o desenvolvimento de aplicações comerciais. Foi desbancada pelo surgimento dos sistemas gerenciadores de bancos de dados. Ainda existem muitos sistemas desenvolvidos nesta linguagem em fase de manutenção. Novas versões contam com Cobol OO.
- Simula é precursora das linguagens orientadas a objetos. Smalltalk é certamente a única linguagem orientada a objetos pura.
- Pascal é a linguagem mais popular de todas as descendentes de Algol. Simples, sistemática e com implementações eficientes foi o carro-chefe da Borland por muitos anos. É uma das linguagens mais recomendadas para ensino de programação. De Pascal surgiram muitas outras linguagens como Modula-2 e Delphi, que passou a incorporar orientação a objetos e definição/manipulação de dados (SQL).
- C é ainda hoje uma das linguagens mais utilizadas, principalmente no ambiente Unix, que foi desenvolvido através dela. Sua marca maior é a eficiência, porém construções de baixo nível tornam seus programas difíceis de serem lidos e depurados. A partir de C, surgiram as linguagens C++ e Java, as quais incorporaram o paradigma de orientação a objetos, porém a segunda eliminou o uso explícito de apontadores e deu ênfase à programação distribuída na Internet, fator primordial de seu sucesso.
- Em outra linha de desenvolvimento, inicialmente voltado para aplicações de inteligência artificial, surgiram as linguagens funcionais e lógicas. Lisp – linguagem funcional – foi a precursora. A linguagem Prolog é largamente utilizada ainda hoje para o desenvolvimento de aplicações cognitivas (novo nome para inteligência artificial). As linguagens funcionais têm ML como representante de maior sucesso. Vale ressaltar também Haskell e Miranda – uma linguagem funcional pura.

I.8 Sintaxe x Semântica

É fundamental para o sucesso de uma LP ter uma descrição concisa e inteligível da linguagem. Os estudos das LP's podem ser divididos em exames de sintaxe e semântica. É importante conhecer os limites entre estes dois conceitos e sua importância:

- **Sintaxe** é o **formato** dos programas – linguagem usada para construir programas, ou seja, é a forma de suas expressões, de suas instruções e de suas unidades de programa.
- **Semântica** – **significado** dos programas – como eles se comportam quando executados em um computador, ou seja, é o significado das expressões, instruções e unidades de programa.

A sintaxe influencia a forma como programas são escritos, enquanto que a semântica dita como os programas devem ser entendidos por outros programadores e executados em um computador.

A semântica é a parte mais importante e que realmente caracteriza uma linguagem. Por exemplo, uma dada construção pode ser apresentada da mesma forma em diferentes linguagens de programação, porém com semânticas diferentes, ou vice-versa. **No final das contas, você só vai saber usá-la adequadamente se conhecer bem o seu significado.** Para corrigir sintaxe, temos os compiladores. Porém desentendimentos semânticos podem nos levar a produzir o programa errado ou com comportamentos indesejados e imprevisíveis.

Exemplo de sintaxe em C:

```
if (<expressão>)  
{  
    <instruções>  
}
```

A semântica dessa forma de instrução obedece à seguinte condição: se o valor atual da expressão for verdadeiro, a instrução incorporada será selecionada para execução.

I.9 Funcionamento das LP's

Um processador de um computador só é capaz de executar um conjunto restrito de operações de hardware muito básicas. Instruções de máquina consistem de vários bytes armazenados na memória principal que instruem o processador a executar uma operação de hardware. *Uma coleção de instruções de máquina na memória principal se chama **programa em linguagem de máquina** ou, como é mais conhecido, **programa executável**.*

Implementar programas em LP's de baixo nível, ou seja, próximo da linguagem de máquina é muito difícil e improdutivo.

A maioria dos programas é criado em alguma LP de alto nível. Nestas linguagens escrevem-se programas utilizando instruções que não são conhecidas pela máquina. O arquivo gerado pelo conjunto de comandos destas LP's é dito **arquivo fonte** ou **código fonte**. Ele não pode ser executado pelo processador.

Para se executar o programa é necessário realizar uma conversão do código fonte para o código de máquina. Isto é feito através de um outro programa: **o tradutor**.

A tradução pode ser feita de duas maneiras: através da **compilação** ou através da **interpretação**.

I.9.1 Compilação

Na compilação todo o programa fonte é traduzido para a linguagem de máquina, gerando-se assim um outro arquivo chamado de programa executável. Este programa pode ser então submetido ao processador e executado pelo mesmo.

O programa que realiza a compilação é chamado de **compilador**.

Vantagens:

- A grande vantagem da compilação é a execução rápida do programa.
- Apenas o código executável é carregado na memória.

Desvantagem:

- Falta de portabilidade.

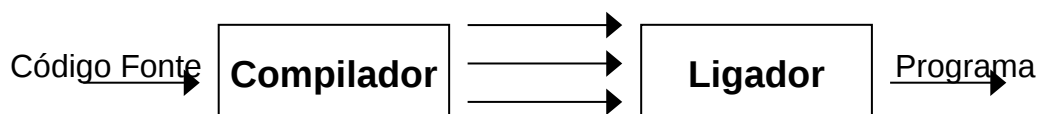


Figura 8: Geração de um código executável

Compilador => converte para código de máquina.

Ligador => Liga as partes de um código, transformando-o em um único bloco.

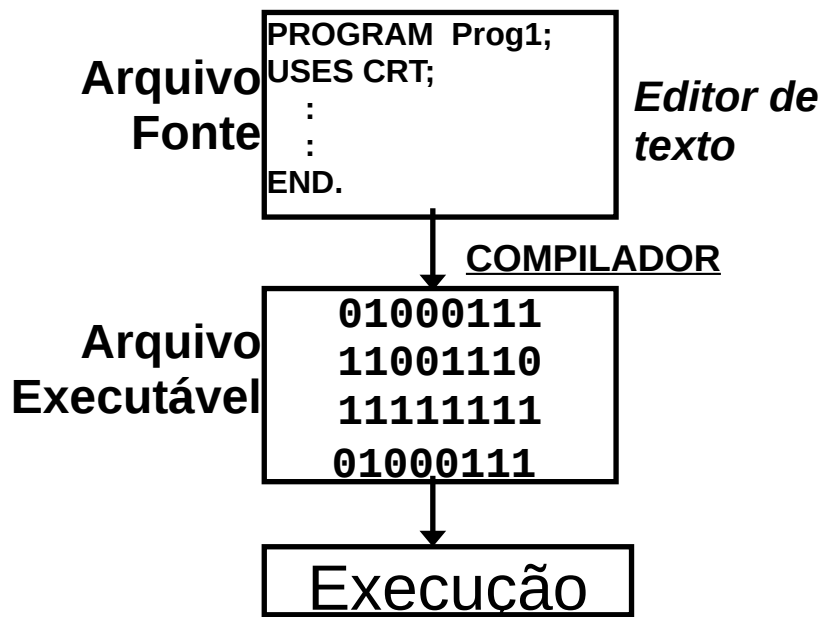


Figura 9: Passos da Compilação

Observe que, a velocidade da conexão entre a memória de um computador e seu processador, normalmente, determina a velocidade do computador, porque, muitas vezes, as instruções podem ser executadas mais rapidamente do que podem ser transferidas para o processador para serem executadas. Essa conexão é chamada de **GARGALO DE VON NEUMANN**. Ele é o principal fator limitante na velocidade da arquitetura de computadores de Von Neumann. O gargalo de Von Neumann foi uma das principais motivações para a pesquisa e desenvolvimento de computadores paralelos.

I.9.2 Interpretação

A interpretação traduz um comando de cada vez, executando-o antes de traduzir o comando seguinte. Não há geração de programa executável.

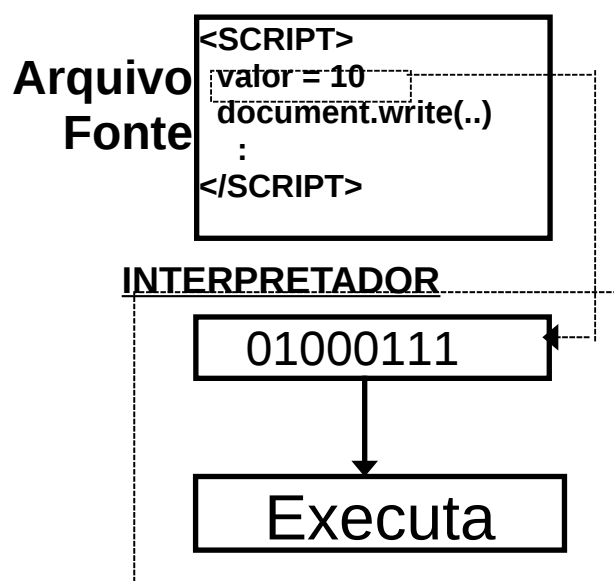


Figura 10: Passos da Interpretação

Vantagem

- Permite fácil implementação de muitas operações de depuração do código-fonte, porque todas as mensagens de erro, em tempo de execução, podem referir-se a unidades de código.
- Portabilidade. Basta que tenhamos um interpretador da LP em cada ambiente e o mesmo programa poderá ser executado em cada um deles.

Desvantagem

- Necessidade de se manter na memória tanto o programa fonte como o interpretador fazendo com que o processo se torne lento.

I.9.3 Mecanismos híbridos

Uma solução para a lentidão da interpretação e a falta de portabilidade da compilação é gerar LP's híbridas, ou seja, LP's que combinem a compilação com a interpretação.

Essas LP's traduzem programas em uma linguagem de alto nível para uma linguagem intermediária, chamada de **bytecode**, projetada para permitir fácil interpretação.

Esse método é mais rápido que a interpretação pura porque as instruções da linguagem fonte são decodificadas somente uma vez.

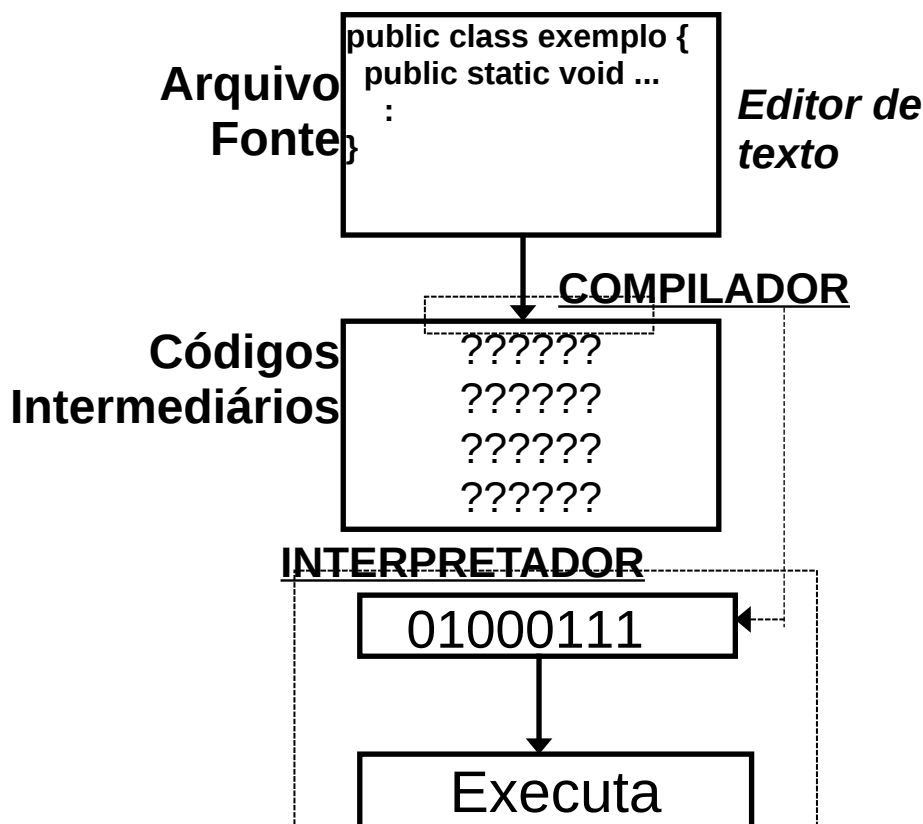


Figura 11: Passos dos Mecanismos Híbridos

I.10 Lista de Exercícios

- 1) Qual a importância de se conhecer os conceitos gerais de linguagens de programação?
- 2) Linguagens de baixo nível são aquelas que estão mais próximas da linguagem de máquina. Quais propriedades desejáveis em LP's as linguagens de baixo nível não satisfariam? Então, por que ainda se utilizam tais linguagens?
- 3) Cite uma vantagem da interpretação e da compilação.
- 4) Por que algumas LPs são híbridas?
- 5) Que linguagem de programação dominou as aplicações comerciais ao longo dos últimos 50 anos?
- 6) Em que linguagem o UNIX foi escrito?
- 7) Qual instrução de controle primitiva é usada para construir instruções de controle mais complexas em linguagens em que elas não existem?
- 8) O que é manipulação de exceções?
- 9) Por que o gargalo Von Neumann é importante?
- 10) Quais os argumentos que você poderia levantar a favor e contra a ideia de uma única linguagem para todos os domínios de programação?
- 11) Avalie o JAVA, usando os critérios descritos neste capítulo.
- 12) Defina, com suas palavras, sintaxe e semântica.
- 13) Explique cada um dos quatro paradigmas e dê uma LP exemplo de cada.

14) Sejam os seguintes fatos escritos em Prolog:

```
progenitor(maria, joao) .  
progenitor(jose, joao) .  
progenitor(maria, ana) .  
progenitor(jose, ana) .  
progenitor(joao, mario) .  
progenitor(ana, helena) .  
progenitor(ana, joana) .  
progenitor(helena, carlos) .  
progenitor(mario, carlos) .  
sexo(ana, feminino) .  
sexo(maria, feminino) .  
sexo(joana, feminino) .  
sexo(helena, feminino) .  
sexo(mario, masculino) .  
sexo(joao, masculino) .  
sexo(jose, masculino) .  
sexo(carlos, masculino) .
```

Quais as respostas das seguintes perguntas?

- a) progenitor(jose, X)
- b) progenitor(X, carlos)
- c) sexo(X, masculino)

- 15) Por que LP's compiladas são mais rápidas?
- 16) Explique o que é portabilidade.
- 17) Dê exemplos de falta de legibilidade em uma LP.
- 18) Explique o que significa uma linguagem ser **flexível**.
- 19) Explique o que significa uma linguagem ser **ortogonal**.
- 20) Cite e explique duas maneiras de como se obter um sistema multiparadigma.
- 21) Escreva a sintaxe do comando for e a semântica do comando switch-case do Java
- 22) Avalie as condicionais abaixo:
- a) (and (>= (+ 1 2) 3) (< 2 5) (- 3 3))
- b) (or (< (+ 1 1) 0) (and (+ 3 1) (> (* 2 2) 5)) (= (- 2 3) -1))
- c) (and (> (* 2 3) (sqrt 81))(or (= (+ 2 3) (* 2 3))(- 3 (* 2 4))))

23) Considere a seguinte função recursiva em LISP:

```
(DEFUN ALGORITM01 (RESULT CONT)
  (IF (<= CONT 0)
    0
    (+ (* RESULT CONT)
      (ALGORITM01 RESULT (- CONT 1)))))
```

Que valor ALGORITM01 deverá retornar se ele for chamado com RESULT=2 e CONT=5?

24) Considere a seguinte função recursiva em LISP:

```
(DEFUN ALGORITM02 (RES CONT)
  (IF (= CONT 1)
    RES
    (* 2 (ALGORITM02 RES (- CONT 1)))))
```

Que valor ALGORITM02 deverá retornar se ele for chamado com RES=3 e CONT=4?

25) Considere a seguinte função recursiva em LISP:

```
(DEFUN MAXIMO (X Y)
  (IF (= Y 0)
    X
    (MAXIMO Y (MOD X Y))))
```

Que valor MAXIMO deverá retornar se ele for chamado com X=40 e Y=25?

26) Considere a seguinte função recursiva em LISP:

```
(DEFUN ALGORITM03 (A B)
  (COND (> A B)
    0)
  (= (MOD A 3) 0)
  (+ A (ALGORITM03 (+ A 1) B)))
  T
  (ALGORITM03 (+ A 1) B))))
```

Que valor ALGORITM03 deverá retornar se ele for chamado com A=1 e B=16?

CAPÍTULO II - A MEMÓRIA

II.1 – Alocação de memória

A memória principal de um computador consiste de uma enorme seqüência contígua e finita de bits. Contudo, a menor unidade que pode ser diretamente endereçada corresponde normalmente ao tamanho da palavra do computador (8 bits, 16 bits, 32 bits, etc..). Podemos imaginar então a memória como sendo um vetor de tamanho finito cujos elementos correspondem ao tamanho da palavra do computador, o qual chamaremos de vetor de memória.

Trataremos aqui a questão da alocação permitida pela LP, ou seja, como a LP trata este tipo de alocação.

Existem basicamente duas formas de reservar memória: **estaticamente ou dinamicamente.**

a) **Estaticamente:** a memória é alocada (reservada) antes que o programa comece sua execução.

Ex. variáveis declaradas em VAR (global) no pascal.

b) **Dinamicamente:** a memória é alocada no momento em que é requisitada, em tempo de execução.

Ex. variável alocada com NEW no pascal.

Cada objeto na memória tem um endereço. Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte. Por exemplo, depois das declarações em Pascal abaixo:

```
c: char;  
i: integer;  
v: array [1..3] of integer;
```

os endereços das variáveis poderiam ser os seguintes:

c	Dseg, 89421
i	Dseg, 89422
v[0]	Dseg, 89426
v[1]	Dseg, 89430
v[2]	Dseg, 89434

II.2 – Ponteiros

Um ponteiro é um tipo especial de variável que armazena apenas endereço de memória. Ele pode ter o valor especial **nil**, que representa um endereço inválido de memória, no caso do Pascal.

Se um ponteiro **p** armazena o endereço de uma variável **i**, podemos dizer **p aponta para i** ou **p é o endereço de i**, ou ainda que **p é uma referência à variável i**. Esquemáticamente, esse apontamento pode ser representado como na figura 12:

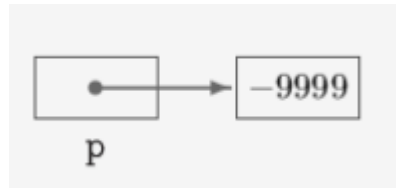


Figura 12: Esquema de um ponteiro

Para uma variável, apontada por um ponteiro, ser utilizada, é necessário que ela seja criada. No caso do Pascal, ela deve ser criada com o uso da palavra **NEW**.

Um ponteiro só pode receber por atribuição um outro ponteiro. Jamais pode receber um inteiro ou qualquer valor de outro tipo. Assim, ele não pode ser lido do teclado nem escrito na tela. Somente o conteúdo da variável que ele aponta que pode ser lido ou escrito.

Por que fala-se que o ponteiro é alocado dinamicamente se no Pascal ou C, por exemplo, ele é declarado na área das variáveis globais?

Por exemplo:

```
PROGRAM      TESTE;
TYPE
  Tp_pont = ^no;
  no = RECORD
    Chave : integer;
    nome  : string[30];
  END;
VAR
  pont : Tp_pont;
```

Na verdade a variável **pont** (neste caso) é alocada estaticamente no segmento de dados. Esta variável conterá apenas um endereço para uma área de memória na heap, onde está localizada a variável **pont^** (dinâmica) que conterá o dado propriamente dito. Esta alocação na heap ocorre quando o comando **new** (no pascal) é utilizado e, só neste momento, que a variável dinâmica será alocada na memória.

Uma estrutura de dados bastante conhecida, que utiliza ponteiros, é a Lista Simplesmente Encadeada (LSE), que é representada pela figura 13:

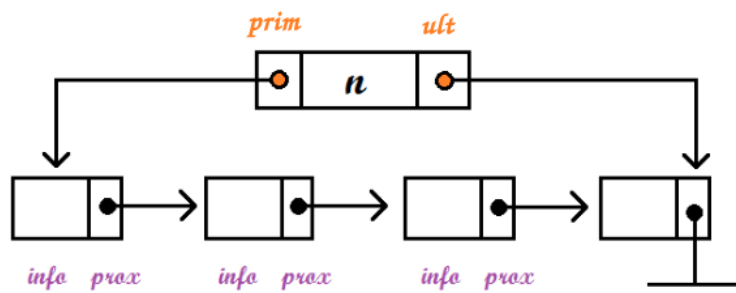


Figura 13: LSE

O ponteiro ***prim*** aponta para o nó inicial e o ***ult*** aponta para o último nó da lista. Cada nó está representado por um retângulo dividido em duas partes. Uma das partes contém a informação e a outra o ponteiro para o próximo nó. Observe que o no último nó a seta aponta para a terra, que indica um endereço inválido de memória, assim a lista chega ao seu fim.

Como ocorre esta alocação?

A forma mais comum de gerenciamento da memória é subdividi-la em áreas.

Uma destas formas é a divisão da memória em 4 segmentos (pascal):

segmento de dados, segmento de pilha, segmento de heap (monte) e segmento de código.

No segmento de código fica armazenado o programa executável.

No segmento de dados ficam localizadas as variáveis estáticas.

No segmento de heap ficam localizadas as variáveis dinâmicas. Apontadas por ponteiros.

No segmento de pilha ficam localizadas variáveis locais, parâmetros e chamadas de procedimentos ou funções.

Neste modelo cada célula de memória é referenciada por um par: **endereço de início do segmento; deslocamento (offset).**

O offset indica a posição da célula de memória dentro do segmento, ou seja, é a distância da célula ao início do segmento.

II.3 Passagem de Parâmetros

Os métodos de passagem de parâmetros são maneiras pelas quais transmitimos um o valor ou damos um caminho de acesso de uma variável para um subprograma. Neste ponto estudaremos dois métodos de passagem de parâmetros:

- ✓ Passagem por valor
- ✓ Passagem por referência

II.3.1 Passagem por valor

A função/procedimento receberá apenas o valor do parâmetro, ou seja, uma variável local será criada no segmento de pilha e o valor que aparece na chamada será atribuído à essa variável local. Assim, quaisquer alterações que sejam feitas no valor do parâmetro, dentro do função/procedimento, não afetarão o valor original, o qual será o mesmo de antes da chamada da função/procedimento. É o método padrão ao declarar os parâmetros de uma função/procedimento em linguagens como Pascal, Java, VB e Delphi é a passagem por valor.

O exemplo abaixo mostra passagem de parâmetro por valor em Pascal. O método *troca* recebe dois inteiros, que são modificados, mas as variáveis originais *num1* e *num2*, declaradas como globais permanecem com seus valores iniciais.

```

1  PROGRAM Passagem_Por_Valor;
2  USES CRT;
3  {-----Declaracao das variaveis}
4  {-----}
5  VAR
6      NUM1, NUM2      : INTEGER;
7  {-----Procedimentos e Funcoes}
8  {-----}
9  PROCEDURE TROCA (A, B : INTEGER );
10 VAR
11     TEMP: INTEGER;
12 BEGIN
13     TEMP:=A;
14     A:=B;
15     B:=TEMP;
16 END;
17
18 {-----Programa Principal}
19 {-----}
20 BEGIN
21     NUM1:=5;
22     NUM2:=10;
23     WRITELN ('VALORES ANTES DA TROCA NUM1 = ',NUM1);
24     WRITELN (' NUM2 = ',NUM2);
25     TROCA (NUM1, NUM2);
26     WRITELN;
27     WRITELN ('VALORES DEPOIS DA TROCA NUM1 = ',NUM1);
28     WRITELN (' NUM2 = ',NUM2);
29     READKEY;
30
31 END.

```

E será mostrado no console:

```

D:\BIN>tpx
Turbo Pascal Version 7.0 Copyright (c) 1983,92 Borland International
VALORES ANTES DA TROCA NUM1 = 5
NUM2 = 10

VALORES DEPOIS DA TROCA NUM1 = 5
NUM2 = 10

```

II.3.2 Passagem por referência

A função/procedimento recebe uma referência ao endereço de memória da variável passada como parâmetro, ou seja, um caminho de acesso, e não uma simples cópia do valor da variável. Ao receber um parâmetro por referência, as alterações que a função/procedimento fizer, serão feitas diretamente na variável original. O parâmetro será então um apelido da variável passada. Esse método oferecido em linguagens como Pascal, C, C++, VB e Delphi.

O exemplo abaixo mostra passagem de parâmetro por referência em Pascal. O método *troca* recebe dois inteiros, que são modificados fazendo com que as variáveis originais *num1* e *num2*, declaradas como globais também tenham seus valores modificados.

```

1  PROGRAM Passagem_Por_Referencia;
2  USES CRT;
3  {-----Declaracao das variaveis}
4  {-----}
5  VAR
6      NUM1, NUM2      : INTEGER;
7  {-----Procedimentos e Funcoes}
8  {-----}
9  PROCEDURE TROCA (VAR A, B : INTEGER );
10 VAR
11     TEMP: INTEGER;
12 BEGIN
13     TEMP:=A;
14     A:=B;
15     B:=TEMP;
16 END;
17
18 {-----Programa Principal}
19 {-----}
20 BEGIN
21     NUM1:=5;
22     NUM2:=10;
23     WRITELN ('VALORES ANTES DA TROCA NUM1 = ',NUM1);
24     WRITELN (' NUM2 = ',NUM2);
25     TROCA (NUM1, NUM2);
26     WRITELN;
27     WRITELN ('VALORES DEPOIS DA TROCA NUM1 = ',NUM1);
28     WRITELN (' NUM2 = ',NUM2);
29     READKEY;
30
31 END.
```

E será mostrado no console:

```

D:\BIN>tpx
Turbo Pascal Version 7.0 Copyright (c) 1983,92 Borland International
VALORES ANTES DA TROCA NUM1 = 5
NUM2 = 10

VALORES DEPOIS DA TROCA NUM1 = 10
NUM2 = 5
```

Java não tem passagem por referência, apesar de no Java os tipos primitivos terem seus valores passados diretamente para as variáveis ao contrário de objetos, cujo valor passado é um endereço de memória, mas mesmo assim, se modificar o endereço do objeto passado, o objeto original não será alterado.

O Java tem uma peculiaridade quando passamos objetos, por exemplo arrays, como parâmetros: Nesse caso, como no Java tudo é referência, o conteúdo do vetor original é

modificado, mas se instanciar novamente o vetor local `vet`, o vetor original não será alterado, ou seja continuará sendo um vetor com duas posições.

```
public class ExemploPassagParametro {
    public static void main(String[] args){
        int num1=10, num2=20;
        troca (num1,num2);
        int[] vetor ={10,20};
        troca (vetor);

    }
    static void troca (int n1, int n2){
        int aux=n2;
        n2 = n1;
        n1 = aux;
    }
    static void troca (int[] vet){
        int aux = vet[0];
        vet[0] = vet[1];
        vet[1] = aux;
        vet = new int[3];
    }
}
```

II.4 Funcionamento da Memória

Com relação ao funcionamento da memória, é necessário fazer algumas observações:

- 1) As variáveis estáticas, que são alocadas no segmento de dados, são alocadas imediatamente antes da execução do programa. Por isso, se não houver espaço em memória para que elas sejam alocadas, **o programa não irá nem compilar**.
- 2) As variáveis locais, que são alocadas no segmento de pilha, são alocadas quando o subprograma em que elas foram declaradas for chamado. Por isso, se não houver espaço em memória para que elas sejam alocadas, ocorrerá **stack overflow** (estouro de pilha) no momento em que o mesmo for chamado.
- 3) As variáveis dinâmicas, armazenadas no segmento heap, são alocadas quando forem instanciadas ou criadas. Por isso, ocorrerá **heap overflow** (estouro de heap) no momento em que ela for instanciada ou criada.

II.5 Exercício

1) Supondo que uma linguagem tenha definido que seu segmento de dados e o segmento de pilha possuem no máximo 64Kbytes cada um e que seu segmento de heap possui no máximo 640Kbytes. Você está implementando um programa nesta linguagem e precisa declarar um vetor de tamanho igual a 10000, onde cada posição armazenará um número real e um caractere. Admitindo-se que um número real ocupa 6bytes de memória e um caractere 1 byte. Perguntas:

- a) Considerando este vetor uma variável global, você terá problemas na declaração? Por que?
- b) Considerando este vetor uma variável local, você teria problemas? Explique.

- c) Se ao invés de vetor você utilizasse alocação dinâmica, haveria problema? Por que?
- d) A implementação de uma variável dinamicamente soluciona o problema de espaço ocorrido com variáveis estáticas ou variáveis locais? Explique.

2) Por que um endereço de memória é composto de duas partes, exemplo: \$435A, \$A45F

3) Suponha a declaração da lista:

PROGRAM EXE;

TYPE

apont = ^NO;

NO =RECORD

nome : string;

prox : apont;

END;

TP_LISTA = RECORD

PRIM, ULT : apont;

END;

VAR

LISTA : TP_LISTA;

A : INTEGER;

AP1, AP2: ^INTEGER;

R : NO;

PROCEDURE TESTE (VAR B:INTEGER);

VAR

L : TP_LISTA;

C, D : apont;

BEGIN

new (C);

new (D);

C^.prox := D;

END;

a) **Responda:** Caso o procedimento seja chamado no programa principal a variável LISTA.PRIM estará alocada em que segmento? E as variáveis A, B e C? A variável apontada por C^.prox está alocada em que segmento?

b) Assinale com um X, no programa principal abaixo, as linhas de código que estiverem corretas.

BEGIN

NEW(LISTA); []

NEW (AP1); []

AP1 := 10; []

AP1^:=10; []

AP2:=AP1; []

WRITELN(AP1); []

NEW (R); []

NEW (R^.PROX); []

NEW (LISTA.PRIM); []

R := LISTA.PRIM; []

LISTA.ULT^:= R; []

END.

Obs.: NEW é um comando usado para alocar variáveis dinâmicas (usado em ponteiros)

Atribuição – somente com variáveis de mesmo tipo.

Leitura e escrita na tela – somente números e caracteres.

4) A memória é dividida em quantas partes? Quais são elas? O que cada uma armazena?

5) O endereço de memória é dividido em quantas partes? Quais são eles?

6) Suponha que uma LP tenha definido seu segmento de dados com 64Kb, o de pilha com 64Kb e o heap com 128Kb. Ao implementar um programa nesta linguagem, é necessário armazenar um vetor de 3.000 posições, onde cada posição armazena um registro contendo uma string 30 caracteres, três inteiros e um real. Admitindo-se que cada caractere ocupa 1 byte, cada inteiro 2 bytes, o real 6 bytes, e cada ponteiro 8 bytes, pergunta-se:

- a) Se este vetor for uma variável global você terá problemas? Por que?
- b) Se o vetor for uma variável local teria problemas? Por que?
- c) Se os elementos forem armazenados em uma lista encadeada, você teria problemas? Por que?
- d) Sugira soluções.

7) Com relação ao programa em Pascal responda às questões abaixo:

PROGRAM EXEMPLO;

TYPE

pont = ^no;

tpinfo = integer;

No = record

info : tpinfo;

prox : pont;

end;

tplista = record

prim, ult : pont;

end;

var

lista : tplista;

ponteiro : pont; {1}

a : integer;

ap1,ap2: ^integer;

R : No;

procedure teste (var a:integer, b : integer);

var

lista : tplista;

C : pont;

begin

...

end;

begin

...

end.

a) Sabendo que um inteiro ocupa 2 bytes de memória e um apontador ocupa 4 bytes, quantos bytes serão alocados ao executar o programa até a declaração {1} ? Justifique mostrando os cálculos.

b) Relacione todas as variáveis do programa, colocando sua localização na memória, se o procedimento TESTE fosse chamado no Programa Principal.

c) Assinale com um X, no programa principal abaixo, as linhas de código que estiverem corretas. Observem que elas não estão necessariamente em sequência.

```
begin
new( lista );           [   ]
new (ap1);              [   ]
R.info := 10;           [   ]
ponteiro := lista^.prox; [   ]
writeln(ap2^);          [   ]
lista.prim^.prox := ponteiro ; [   ]
new (R);                [   ]
teste (R.info, ap1^);   [   ]
ap1 := a;               [   ]
R^.info := 10 ;        [   ]
end.
```

8) Com relação à memória, qual a diferença de passagem de parâmetro por valor e por referência?

9) Um vetor bi-dimensional $A[1..lin, 1..col]$ de inteiros (2 bytes) é armazenado na memória, começando no endereço S. Monte uma expressão matemática que é capaz de mostrar a correta posição de um elemento arbitrário $A[i,j]$?

CAPÍTULO III - VALORES E TIPOS

Dados são a matéria prima da computação, pois os programas produzem resultados manipulando-os. Por isso é importante que uma linguagem suporte vários tipos de dados e estruturas. E, são esses tipos que permitem a representação de valores nos programas.

III.1 – Valores

Tudo que pode ser avaliado, armazenado, incorporado a uma estrutura de dados, passado como argumento para um procedimento ou função, retornado como resultado de funções, etc.

III.2 – Tipos

Tipos são grupos de valores que exibem comportamento uniforme perante as mesmas operações.

Podemos definir Cardinalidade de um tipo (#) como o número de valores distintos que o tipo pode assumir.

Exemplos:

#booleano = 2 (em pascal) #inteiro = limite inferior + limite superior + 1

#inteiro = limite inferior – limite superior + 1

A quantidade de tipos de dados suportada depende da LP. As LP's podem permitir a criação de tipos de dados do usuário, bem como a criação de estruturas de tipos de dados.

O tipo de dado de uma variável está registrado em um descritor. O descritor da variável consiste no conjunto dos seguintes atributos:

- ✎ **Nome:** identificador (símbolo), que serve para referenciar o dado.
- ✎ **Endereço:** posição na memória onde está localizado o dado.

Os tipos de dados se dividem em Tipos primitivos e Tipos compostos ou estruturados.

III.2.1 Tipos Primitivos

➔ Definição:

- São conjunto de valores que não podem ser decompostos em valores mais simples.
- Tipos de dados que não podem ser definidos em termos de outros tipos.

➔ Características importantes

- Formam, juntamente com um ou mais tipos primitivos, os tipos estruturados.
- Estão associados com o hardware.
- Os tipos primitivos de uma linguagem dependem de sua área de aplicação:
 - LP comercial COBOLDecimal
 - LP numéricaFORTRAN números reais com escolha de precisão

→ Tipo primitivo discreto = é aquele que possui uma relação 1 pra 1 com um intervalo dos inteiros. Exemplo: caractere

III.2.1.1 Inteiro

O tipo de dados primitivo mais comum é o inteiro. Algumas linguagens permitem diversos tamanhos de inteiros.

Exemplos:

PASCAL :	BYTE	= 0 a 255	1 byte
	SHORTINT	= -128 a 127	1 byte
	WORD	= 0 a 65535	2 bytes
	INTEGER	= -32768 a 32767	2 bytes
	LONGINT	= -2.147.483.648 a 2.147.483.647	4 bytes

Java :	byte	= -128 a 127	1 byte
	short	= -32768 a 32767	2 bytes
	int	= -2.147.483.648 a 2.147.483.647	4 bytes
	long	= -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807	8 bytes

Um valor inteiro pode ser representado em um computador por uma string de bits, com um dos bits mais a esquerda representando o sinal do número.

III.2.1.2 Ponto-flutuante

É o armazenamento de um número real no formato científico sem delimitar limite superior e inferior. Como a representação é finita, números reais com dízimas periódicas e números especiais (π por exemplo) só podem ser representados por aproximações.

Exemplo: $3,14 = 3,14 \times 10^0$
 $0,000001 = 1,00 \times 10^{-6}$
 $1941 = 1,941 \times 10^3$

O armazenamento em ponto flutuante economiza espaço de memória. A representatividade é maior.

Em C:

float = seis dígitos de precisão	ocupa 4 bytes
double = dez dígitos de precisão	ocupa 8 bytes

O tipo ponto-flutuante segue o Padrão de Ponto Flutuante IEEE 754. Este padrão define três formatos: precisão simples (32 bits); precisão dupla (64 bits) e precisão estendida (80 bits).

simples = intervalo de expoente entre -126 e 127
dupla = intervalo de expoente entre -1022 e 1023

III.2.1.3 Decimais

Esse tipo de dado armazena um número fixo de dígitos decimais, com o ponto decimal em uma posição fixa do valor. São utilizados para processamento de dados comerciais, sendo fundamental para o COBOL.

Os tipos decimais têm a vantagem da capacidade de armazenar, com precisão valores decimais, pelo menos os de dentro de uma faixa restrita, o que não pode ser feito em ponto flutuante.

III.2.1.4 Booleanos

Assumem apenas dois valores: verdadeiro e falso, por isso formam os tipos mais simples.

Normalmente são usados como *flags* (sinalizadores) em programas, aumentando a legibilidade. C é uma exceção, onde qualquer inteiro diferente de zero é considerado verdadeiro e o zero é considerado falso.

Apesar de poderem ser representados por apenas um bit, geralmente ocupam 1 byte de memória por questões de facilidade de acesso.

III.2.1.5 Caractere

Os caracteres são armazenados nos computadores como codificação numérica. A mais utilizada é a ASCII (American Standard Code for Information Interchange), que usa valores de 0 a 127 para codificar 128 caracteres diferentes.

Uma outra tabela de código foi desenvolvida: a UNICODE (16 bits), pois com a globalização a tabela ASCII está tornando-se obsoleta. Esta nova codificação inclui o alfabeto cirílico, usado na Sérvia, e os dígitos tailandeses, por exemplo. O Java é a primeira linguagem amplamente usada a adotar o conjunto de caracteres UNICODE.

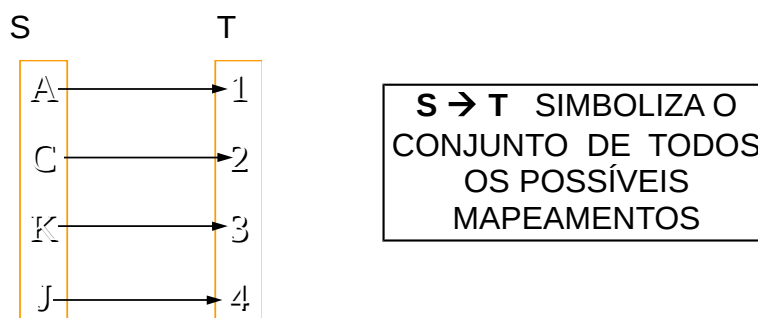
III.2.2 Tipos compostos

É um tipo composto a partir de tipos mais simples.

Exemplos: registro, vetor, strings, etc.

III.2.2.1 Mapeamentos

Consiste no mapeamento de um conjunto de valores para outro (que pode ser do mesmo tipo)



Mapeamento finito = Funções cujo domínio é finito.

Podemos observar que vetores são um exemplo, pois há um mapeamento finito do conjunto índice para o conjunto dos componentes.

III.2.2.2 Vetor

- ✎ Servem para criar um agregado homogêneo de elementos de dados, ou seja, todos os valores de dados têm o mesmo tipo e são processados da mesma maneira.
- ✎ Cada elemento é acessado por um índice.
- ✎ O conjunto índice deve ser finito e discreto.
- ✎ A maior parte das LP's restringe o conjunto índice a um intervalo dos inteiros. Algumas fixam o limite inferior (C, C++, Java) outras não.
- ✎ Pascal e ADA permitem que os índices sejam de qualquer tipo primitivo discreto (inteiro, enumerado ou intervalo).
- ✎ Dependendo da linguagem a estrutura pode ser manipulada como um todo. Porém, linguagens como Ada e Algol-68 permitem a seleção de um sub-array.
- ✎ Em geral, o uso incorreto de índices (ex. acessar um índice inexistente) só pode ser verificado em tempo de execução.
- ✎ Linguagens como Pascal, ADA e Java fazem verificação dinâmica dos índices (aumenta confiabilidade, mas reduz eficiência)
- ✎ Linguagens como C, C++ e FORTRAN não fazem verificação de índices.

Exemplo de problema que pode ocorrer em C:

```
int v[7];  
v[13] = 198; // atribuição válida com consequências imprevisíveis
```

☺ **Quantidade de índices de um vetor depende da linguagem.**

<code>FORTRAN 1</code>	<code>= 3</code>
<code>PASCAL</code>	<code>= limite de memória</code>

Devemos observar que os elementos de um vetor de uma dimensão são armazenados de forma contígua na memória. Vetores de mais de uma dimensão são normalmente armazenados como vetores de uma dimensão de tamanho igual ao número de células do vetor multidimensional. Por exemplo, os vetores bidimensionais costumam ser armazenados por linha.

Assim, o acesso ao elemento de um vetor multidimensional é bastante custoso, pois necessita de um cálculo da posição do elemento na memória. Se for preciso uma aplicação utilizando o máximo de eficiência computacional, talvez seja melhor substituir uma representação multidimensional por outra unidimensional.

Normalmente, em um vetor multidimensional o número de elementos de cada dimensão é fixo. Mas, em Java, isto pode não acontecer, pois os vetores multidimensionais são, na verdade, vetores unidimensionais, onde cada elemento é outro vetor independente. Por exemplo: um vetor bidimensional é considerado um vetor unidimensional onde todos os

elementos são vetores unidimensionais independentes e que, portanto, poderão ser criados com tamanhos diferentes.

A declaração abaixo cria um vetor bidimensional com 5 linhas onde a primeira linha tem 1 coluna, a segunda linha tem 2 colunas, e assim por diante.

```
int[][] vet = new int[5][];  
for (int i=0; i<vet.length; i++)  
    vet[i] = new int[i+1];  
}
```

☺ **Inicialização de vetores**

Algumas linguagens, como C, C++, Java, FORTRAN 77, permitem a inicialização na declaração. Mas outras, como PASCAL e MODULA-2, não permitem na seção de declaração de variáveis.

Exemplo: C `int lista [] = {4, 5, 7, 83}`

☺ **Operações com vetores**

As operações, que podem ser realizadas com vetores, dependem da linguagem.

APL, por exemplo, possui diversos comandos para se trabalhar com vetores e matrizes (as quatro operações aritméticas básicas, inverter a matriz, inverter a coluna, etc).

PASCAL permite utilizar um comando de atribuição entre dois vetores: `V1 := V2;`

☺ **Categorias de um vetor**

Existem quatro categorias de vetores:

- ✎ Estáticos
- ✎ Semi-estáticos
- ✎ Semi-dinâmicos
- ✎ Dinâmicos

✎ **Estático**

É aquele em que os índices são definidos em tempo de compilação e a alocação de memória é estática, ou seja, realizada antes da execução. Seu tamanho é fixo e seus elementos ficam armazenados no Segmento de Dados.

Vantagem ⇒ eficiência de tempo.

Exemplo: FORTRAN 77, declaração global de um vetor no PASCAL

✎ **Semi-estático (stack-dinâmico fixo)**

É aquele em que os índices são definidos em tempo de compilação e a alocação de memória é dinâmica, ou seja, realizada durante a execução. Seu tamanho é fixo e seus elementos ficam armazenados no Segmento de Pilha, sendo, portanto, variáveis locais. Assim, a memória só será alocada ao chamar o procedimento ou função em que o vetor for declarado.

Vantagem ⇒ eficiência de espaço.

Exemplo: Pascal, C (a declaração dos procedimentos e funções)

✎ Semi- dinâmico (stack-dinâmico)

É aquele em que os índices são definidos em tempo de execução e a alocação também. Porém, após a especificação dos limites não se pode mais alterá-los. Seus elementos podem ser armazenados no segmento de Pilha. Assim, a memória só será alocada quando forem definidos os índices do vetor.

Vantagem ⇒ flexibilidade, ou seja, o tamanho de um vetor não precisa ser conhecido até que ele seja usado.

Exemplo: **ADA**

```
GET (LIST_LEN);  
declare  
  LIST: array (1..LIST_LEN) of INTEGER;  
begin  
  ...  
end;
```

Em declare é alocado e em end; é desalocado.

✎ Dinâmico (heap-dinâmico)

É aquele em que os índices e a alocação da memória ocorrem dinamicamente e podem ser modificados a qualquer momento. Este tipo de vetor fica alocado no Segmento Heap. O aumento ou diminuição do vetor são implementados por meio da alocação de um novo espaço de memória para o vetor, da cópia do conteúdo (se for o caso) e da desalocação da memória que tinha sido reservada para ele.

Exemplo: FORTRAN 90

```
INTEGER, ALLOCATABLE, ARRAY(:, :) :: MAT
```

A linha acima é a declaração de um tipo matriz MAT de elementos do tipo inteiro que só será alocada dinamicamente pela instrução ALLOCATE.

```
ALLOCATE (MAT( 10, NUMERO_DE_COLS))
```

Para desalocar o vetor, é necessário usar o comando:

```
DEALLOCATE (MAT).
```

C e C++ também permitem vetores dinâmicos utilizando **malloc** e **free**, **new** e **delete**.

C:

```
void f (int a){  
  int *vet;  
  vet = (int *)malloc(a * sizeof(int));  
  vet[0] = 10;  
  free (vet);  
}
```

C++

```
void f (int a){  
  int *vet;  
  vet = new int[a];  
  vet[0] = 10;  
  delete[] vet;  
}
```


JAVA:

```
int[] criaVetor (int a){  
    int[] vet;  
    vet = new int[a];  
    vet[0] = 10;  
    return vet;  
}
```

É possível desalocar o vetor a qualquer instante:

```
vet = null;
```

A implementação de vetores dinâmicos e semi-dinâmicos em C ou C++ é complexa e bastante suscetível a erros, pois é tarefa do programador gerenciar a alocação e desalocação de memória. Por outro lado, Java cria programas mais legíveis e confiáveis, já que os vetores são criados sem a utilização explícita do conceito de ponteiros e o programador não precisa desalocar explicitamente a memória que acabou de utilizar.

III.2.2.3 String

A string na verdade é uma cadeia de caracteres. São usadas para armazenar dados não numéricos e para entrada e saída de dados.

Algumas LP's podem considerar uma String como tipo primitivo (PERL, SNOLBOL e ML), pois fornece comandos para se trabalhar com a String automaticamente, não sendo necessário que o usuário implemente coisa alguma.

Outras LP's tratam Strings como um mapeamento finito, ou seja, como um vetor de caracteres (C, C++, Pascal e Ada). As operações que manipulam Strings são as mesmas que manipulam vetores e/ou são criadas bibliotecas com funções próprias para Strings.

Existem ainda aquelas que consideram uma String como uma lista recursiva (MIRANDA, PROLOG e LISP). Estas linguagens têm o tipo lista como um tipo predefinido.

Ex.:

ADA ⇒ String é um array unidimensional de **CHARACTER**, que apresenta operações como referências a substrings, concatenação, operadores relacionais e atribuição.

C/C++ ⇒ String é um array unidimensional de **char**. Há uma biblioteca padrão (**string.h**) que fornece operações com este tipo de dado, como cópia de string (**strcpy**), comparação (**strcmp**), tamanho da string (strlen), etc.

Pascal ⇒ String é um array unidimensional de **char**. As operações de cópias e comparação são feitas através dos operadores de atribuição e relacionais, respectivamente. Já operações como a concatenação pode ser feita através do comando **concat** ou do operador **+**.

Java ⇒ String é considerada um novo tipo (uma classe da biblioteca padrão). Assim, ela possui suas próprias operações.

Com relação ao tamanho de uma String, há três formas de implementação:

✎ **Estática:** O tamanho da string é predefinido ou definido na compilação e não pode ser modificado durante a execução. É o que acontece, por exemplo, no FORTRAN 90, COBOL, PASCAL e ADA. Se, no Pascal, uma variável **nome**, declarada com tamanho

30 (declaração abaixo), for preenchida com 20 caracteres, os 10 restantes serão preenchidos com caracteres brancos (espaços vazios).

```
nome: string[30]; {Pascal}
```

- ✎ **Dinâmica Limitada:** O tamanho da string é variável até um máximo fixo e definido na declaração da variável, ou seja, a string pode armazenar qualquer quantidade de caracteres entre 0 e o máximo. Podemos tomar como exemplo o C e o C++. Se, em C/C++ uma variável **nome** for declarada como abaixo, é criado um ponteiro que aponta para a cadeia de caracteres, **aluno0**, onde **0** é o caractere nulo que indica o final da string. Portanto, estas linguagens armazenam um caractere especial para indicar o final da string, ao invés de armazenarem seu tamanho.

```
char *str = "aluno";
```

- ✎ **Dinâmica:** O tamanho da string é variável e sem nenhum máximo. Exemplos são as Strings de PERL, SNOLBOL, APL e JAVASCRIPT. Nestas LP's, sempre ocorre alocação e desalocação de memória quando há necessidade de aumentar uma string.

III.2.2.4 Tipo Ponteiro

Um tipo ponteiro é aquele em que as variáveis têm uma faixa de valores que consistem em endereços de memória e um valor especial, o NIL (não é um endereço válido, serve apenas para indicar que o ponteiro não pode ser utilizado para referenciar qualquer célula de memória).

Um ponteiro pode ser usado para acessar uma área de memória em que o armazenamento é alocado dinamicamente.

Eles não são tipos estruturados nem variáveis escalares pois, são mais usados para referenciar uma variável e não armazenar um dado.

Exemplo: Pascal (^), C e C++ (*) e Ada (access).

```
TYPE
  ponteiro = ^no;
  no      = RECORD
    item : Tp_item;
    prox : ponteiro;
  END;
VAR
  P, Q : ponteiro;
```

Em ML qualquer tipo: `DATATYPE intlist = nil`
`cons of int*intlist`

Java não usa ponteiros explicitamente, mas existem ponteiros sob a superfície como forma de endereços de memória. Vejamos o exemplo abaixo:

```
int num;
Data dt1;
```

Ao declararmos uma variável do **int**, um local da memória chamado **num** mantém um valor inteiro. Entretanto, o local da memória **dt1** não mantém os dados do objeto **Data**,

mas sim o endereço de um objeto **Data** que é realmente armazenado em outro lugar da memória. Assim, o nome **dt1** é um referência ao objeto Data. E não o próprio objeto.

Vantagem: usa memória dinâmica (heap) e apresentam poder de endereçamento direto.

Operações com ponteiros

🐼 **Atribuição** ⇒ pode ser feita entre variáveis do tipo ponteiro ou entre uma variável ponteiro e um endereço de memória. Por exemplo, em C:

Em C:	Em Pascal:
// dois ponteiros para int e	P, Q: ^INTEGER;
// um int	R : INTEGER;
int *p, *q, r;	BEGIN
// atribui o endereço da	NEW(P);
// variável r para q	...
// atribui o endereço armazenado	P^:=R;
// em q a p	Q:=P;
q = &r;	...
p = q;	END.

🐼 **Alocação** ⇒ armazena dinamicamente um espaço de memória acessado via ponteiros e retorna o endereço da primeira célula alocada.

Em C:	Em Pascal:
int *p = (int*) malloc (sizeof(int));	P, Q: ^INTEGER;
	R : INTEGER;
	BEGIN
	NEW(P);
	...
	End.

🐼 **Desalocação** ⇒ força a liberação de áreas de memória que estão alocadas e apontadas pelo ponteiro.

Em C:	Em Pascal:
free(p);	dispose (p);

🐼 **Desreferenciamento** ⇒ retorna o conteúdo do que é apontado pelo ponteiro. Pode ser implícita como Java ou explícita como C ou Pascal:

Em C:	Em Pascal:
int *p;	P : ^INTEGER;
*p = 10;	BEGIN
*p = *p + 10;	NEW(P);
	P^ = 10;
	P^= P^ + 10;
	END.

Em Java:
String str = new String("eu");
str += " e você";

Problemas com ponteiros

A programação com ponteiros requer muita atenção do programador para não gerar erros durante sua manipulação. Os problemas mais comuns que ocorrem são:

🐭 **Baixa Legibilidade** ⇒ Ponteiros são conhecidos como o GOTO das estruturas de dados.

Portanto, o programador deve ter muita disciplina e cuidado. Por exemplo, ao fazer:

```
P.ITEM.PROX^ := Q;
```

Não podemos, rapidamente verificar a estrutura que está sendo atualizada.

🐭 **Erro de Violação do Sistema de Tipos** ⇒ Em LP's nas quais o tipo apontado pelo ponteiro não é restrito, expressões contendo ponteiros podem ser avaliadas com valores de tipos diferentes do esperado originalmente, gerando erros na avaliação em tempo de execução. Por exemplo, em Pascal:

```
P: ^ITEM;  
R : ^INTEGER;  
BEGIN  
    ...  
    R := P;  
    Writeln (R);  
    ...  
END.
```

Se ITEM for um registro , p. ex., neste caso, o valor exibido na tela é imprevisível, pois R agora aponta para um registro e não para um inteiro.

🐭 **Ponteiro Oscilante** ⇒ contém um endereço de uma variável dinâmica desalocada.

Exemplo:

```
P:=Q;  
Dispose (Q);
```

a) a localização para a qual ele aponta pode ter sido realocada para outra variável. Se forem tipos diferentes ⇒ ponteiro inválido.

b) a posição pode estar sendo utilizada pelo SO para alguma operação interna (temporariamente).

🐭 **Variáveis Heap-Dinâmicas Perdidas** ⇒ variáveis dinâmicas que não estão mais acessíveis. São chamadas de **lixo** porque não são mais úteis, já que não podem mais ser acessadas, nem podem ser realocadas novamente para outro uso.

Exemplo:

Uma lista com um único apontador P apontando para o primeiro elemento. Se o programador fizer:

```
P := P^.prox;
```

Perdeu o primeiro elemento.

Obs. PASCAL aponta somente para dentro da heap. C pode apontar, virtualmente, para qualquer variável em qualquer lugar da memória. Java não tem ponteiro oscilante.

Variáveis desse tipo:

- a) Deixaram de ser úteis, não podem mais serem acessadas.
- b) não podem ser realocadas para um novo uso.

Soluções para ponteiros oscilantes ⇒ ler Sebesta – capítulo 6 – 6.10.10.2 – página 258-259

III.2.2.5 Registro

Um registro é um agregado possivelmente heterogêneo de elementos que são identificados por nome. Seus diferentes campos não são processados da mesma maneira, como também seus campos não precisam ser acessados em uma ordem sequencial.

A referência a campos pode ser de duas formas: **Referência Amplamente Qualificada e Referência Elíptica.**

Referência Amplamente Qualificada

É aquela em que o nome do registro e o campo do registro, para ser acessado, necessitam de ser especificados.

Exemplo (no pascal): Supondo um tipo registro definido como no exemplo abaixo:

```
TYPE
  Tp_reg = RECORD
      Nome:STRING;
      Serie : STRING[5];
  END;
```

A variável deste tipo deve ser declarado como:

```
VAR
  R : Tp_reg;
```

Então, o campo nome deve ser referenciado da seguinte maneira:

```
BEGIN
  Writeln ('Digite o nome:');
  Readln (R.Nome);
  Writeln('Digite a serie');
  Readln (R.Serie);
  END;
:
END.
```

Referência Elíptica

O nome do registro pode ser omitido e apenas os campos seriam especificados.

Exemplo (no pascal):

```
BEGIN
  WITH R DO
    BEGIN
      Writeln ('Digite o nome:');
      Readln (Nome);
      Writeln('Digite a serie');
      Readln (Serie);
    END;
  :
END.
```

- Sua cardinalidade é: $\#(\text{registro}) = \# \text{tipo_campo1} * \# \text{tipo_campo2} * \dots * \# \text{tipo_campo_n}$

III.2.2.6 União disjunta

É um tipo que pode armazenar diferentes valores de tipo durante a execução do programa.

Exemplo: você precisa de uma variável que seja capaz de armazenar tanto um inteiro quanto um caractere.

Nas declarações normais isto não seria possível. Utiliza-se então a idéia de união disjunta, que em tempo de execução reserva o espaço apropriado para o valor a ser armazenado.

PASCAL, ADA = Registro variante

C, C++, Algol 68 = union

COBOL = redefines

ML = constructions

Exemplo em PASCAL – registro variante:

TYPE

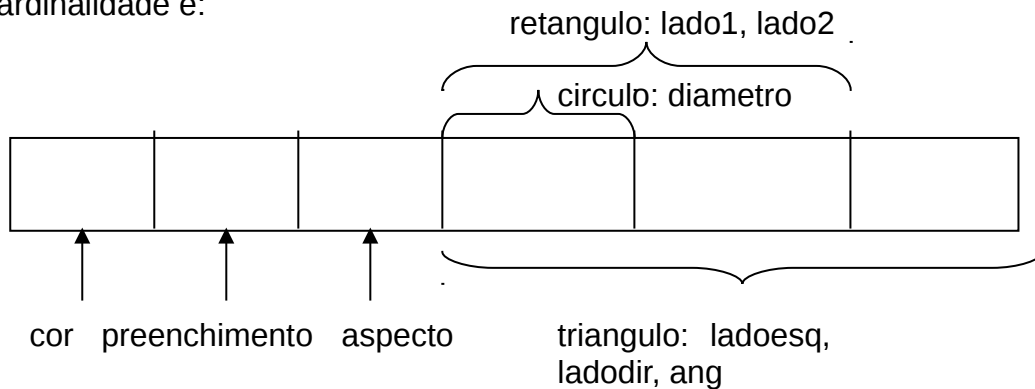
```
Forma = (circulo, triangulo, retangulo);
Cores = (vermelho, verde, azul);
Figura = record
  Preenchido : Boolean;
  Cor : cores;
  case aspecto : forma of
    circulo : (diametro : real);
    triangulo : (ladosq : integer; ladodir : integer; angulo : real);
    retangulo : (lado1: integer; lado2 : integer);
  END;
END;
```

VAR

```
Minhafigura : figura;
BEGIN
  ...
  case minhafigura.aspecto of
    circulo :
      writeln ('O diametro do circulo = ', minhafigura.diametro);
    triangulo :
      BEGIN
```

```
writeln ('Os lados do triangulo sao = ', minhafigura.ladoesq,
        minhafigura.ladodir);
writeln (0 angulo entre os lados e = ', minhafigura.angulo);
END;
retangulo :
    writeln ('Os lados do retangulo sao = ', minhafigura.lado1,
            minhafigura.lado2);
END;
...
END.
```

Sua cardinalidade é:



Em ADA, teremos:

```
Type FORMA is (CIRCULO, TRIANGULO, RETANGULO);
Type CORES is (VERMELHO, VERDE, AZUL);
Type FIGURA (ASPECTO : FORMA) is
    Record
        PREENCHIDO : BOOLEAN;
        COR : CORES;
        Case ASPECTO is
        When CIRCULO =>
            DIAMETRO : FLOAT;
        When TRIANGULO =>
            LADOESQ : INTEGER;
            LADODIR : INTEGER;
            ANGULO : FLOAT;
        When RETANGULO =>
            LADO1 : INTEGER;
            LADO2 : INTEGER;
        END case;
    END record;
```

As duas instruções a seguir declaram variáveis do tipo figura:

```
FIGURA_1 : FIGURA;
FIGURA_2 : FIGURA (ASPECTO +. TRIANGULO);
```

FIGURA_1 é um registro variante não-restringido que não tem valor inicial. Seu tipo pode mudar pela atribuição. Por exemplo:

```
FIGURA_1 := (PREENCHIDO => true,  
              COR => AZUL,  
              ASPECTO => RETANGULO,  
              LADO_1 => 12,  
              LADO_2 => 3);
```

A variável FIGURA_2 é restringida para ser um triângulo e não pode ser mudada para outra variante.

OBERON, Modula-3 e Java = Não permitem uniões (segurança).

III.2.2.7 Tipos Enumerados

É aquele em que os valores se tornam constantes simbólicas. Possuem uma relação direta com os inteiros na ordem em que aparecem. Isto é, um conjunto de constantes é representado pelos identificadores. Vejamos como trabalhar com um tipo enumerado:

Em Pascal:

- Não é permitido que um valor participante de um tipo enumerado componha outro tipo.
- A variável do tipo enumerado não pode receber valores lidos do teclado nem seu conteúdo pode ser exibido via comando write (exibição no Pascal),
- Como possuem uma relação direta com os inteiros, variáveis destes tipos podem ser usadas em comparações. No exemplo dado em Pascal (acima) poder-se-ia fazer **IF (Sexo < feminino) THEN**, esse comando retornaria um valor verdadeiro.

Exemplo:

TYPE

```
Tp_sexo = (masculino, feminino);  
Tp_semana = (segunda, terça, quarta, quinta, sexta, sábado,  
             domingo);
```

VAR

```
Sexo : Tp_sexo; {variável Sexo só assume o valor masculino ou  
                feminino}  
I: Tp_semana;
```

BEGIN

```
Sexo := 1; //erro pois Sexo é do tipo enumerado e não inteiro  
Sexo := masculino;  
IF (Sexo < feminino) THEN  
FOR I:= segunda TO domingo DO  
BEGIN  
....  
END;
```

CASE Sexo OF

```
masculino : ...  
feminino : ...
```

END;

END.

Em C:

- ✓ A uma variável de tipo enumerado podem ser atribuídas apenas as constantes definidas na declaração do tipo enumerado.
- ✓ Internamente, o compilador designa o valor 0 para o primeiro símbolo da enumeração, e incrementa de um o valor associado a cada símbolo na sequência. Isto pode ser modificado se o programador quiser através de atribuição explícita de um valor inteiro a um símbolo, como nos exemplos abaixo.

Sintaxe:

```
enum [<nome>] (<ident1>[=<const1>], [<ident2>[=<const2>],] ... );
```

Elemento	Descrição
nome	Nome do tipo enumerado
identN	Nome do identificador da constante
constN	Valor da constante. Se não for especificado, será o valor da constante anterior acrescido de 1 ou zero caso seja const1

Exemplo:

```
#include <stdio.h>
#include <conio.h>
```

```
void main(){
    enum Sexo {MASCULINO, FEMININO};
    //Neste exemplo, MASCULINO corresponde a 0 e FEMININO a 1

    enum Mes {JAN=1, FEV, MAR, ABR, MAI, JUN, JUL, AGO, SET, OUT,
              NOV, DEZ};
    //Agora, JAN corresponde a 1, FEV a 2 e assim por diante, pois
    //foi atribuído um valor inicial à primeira constante.
```

As variáveis são declaradas com o tipo, seguido do nome:

```
Sexo sex;
Mes meses;
```

A variável também pode ser declarada seguida de um tipo sem nome:

```
enum {calculo=3, computacao=8, fisica, criptografia=15,algebra}
    materia;
printf("%i,%i,%i,%i,%i", calculo, computacao, fisica, criptografia,
        algebra); //imprime os valores

for (meses =jan; meses < jul; meses++){...}
if (sex == MASCULINO) {...}
if (meses == DEZ)
    meses = JAN;
else
    meses++;
}
```

Em Java o tipo enumerado só apareceu na versao 1.5 (Java 5.0). Este tipo é um pouco mais complexo por que é este tipo é uma classe, onde é possível ao programador criar

sua própria classe de tipo enumerado. Vejamos abaixo, o uso comum de uma objeto do tipo enum.

```
public class TesteEnum {;
    // enumeração com constantes que representam o sexo
    public enum TpSexo {MASCULINO, FEMININO};
    public static void main (String[] args){
        //definição de um objeto que pode conter MASCULINO ou FEMININO
        TpSexo sexo;

        sexo = TpSexo.MASCULINO;
        switch (sexo){
            case MASCULINO :
                System.out.println (sexo);//MASCULINO
                System.out.println(sexo.ordinal());//0
                System.out.println(sexo.toString());//MASCULINO
                break;
            case FEMININO :
                //...
        }
    }
    //0 for será rodado do início até o fim dos elementos do tipo enumerado.
    for (TpSexo sex : TpSexo.values()){
        System.out.println (sex);
        System.out.println(sex.ordinal());
    }
}
```

Observações sobre o tipo:

- Para que a LP permita o uso de valores iguais em tipos enumerados diferentes é necessária uma verificação maior.
- Como possuem uma relação direta com os inteiros, variáveis destes tipos podem ser usadas em comparações. No exemplo dado em Pascal (acima) poder-se-ia fazer **IF (Sexo < feminino) THEN** , esse comando retornaria um valor verdadeiro.

III.2.2.8 Tipos Sub-faixa

É um tipo estabelecido usando-se como valores uma sub-faixa de outro tipo.

Exemplo (pascal):

```
TYPE
    Tp_maiúscula = 'A' .. 'Z';
    Tp_numeros = 1..10;
VAR
    C : Tp_maiúscula;
BEGIN
    READLN ( C );
END.
```

Pascal apresenta um problema: a variável C aceitará qualquer valor do tipo caractere independente de estar fora do limite -> erro de verificação de tipos.

III.2.2.9 Tipo Conjunto

É um tipo definido por uma coleção de valores distintos todos do mesmo tipo. No conjunto, não há noção de ordem dos elementos. As operações mais importantes de um tipo conjunto são:

- ✎ Adicionar elementos no conjunto (descartando duplicações)
- ✎ Remover um elemento pertencente ao conjunto
- ✎ Acessar elementos
- ✎ Pesquisar elementos (descobrir se um elemento pertence ou não ao conjunto)
- ✎ Obter a quantidade de elementos

Exemplo (**PASCAL**):

- A variável do tipo enumerado não pode receber valores lido do teclado ou seu conteúdo ser exibido via comando write.

TYPE

```
Tp_semana = (seg, ter, qua, qui, sex, sáb, dom);  
Tp_conj_semana = set of Tp_semana;
```

VAR

```
Semana, s1, s2: Tp_conj_semana;
```

BEGIN

```
Semana := []; {conjunto vazio}  
Semana := [segunda];  
S1 := [domingo];  
S2:= semana + s1;
```

END.

Com uma variável do tipo conjunto é possível realizar as seguintes operações: união, interseção, diferença, comparações e teste de presença.

No Pascal temos:

```
+ ⇒ união  
* ⇒ interseção  
- ⇒ diferença  
<= ou >= ⇒ teste de inclusão  
= ou <> ⇒ teste de igualdade  
in ⇒ teste de presença
```

Exemplo (**JAVA**):

```
import java.util.Set;  
import java.util.HashSet;  
public class Conjunto {  
    public static void main (String[] args){  
        Set<String> conj = new HashSet<String>();  
        Set<String> conj2 = new HashSet<String>();  
        Object[] itens;  
  
        conj.add("a");  
        conj.add("b");  
        conj.add("c");
```

```
conj.add("b");
conj2.add("a");
conj2.add("nome");
System.out.println ("a está "+conj.contains("a"));
System.out.println ("b está "+conj.contains("b"));
System.out.println ("b está "+conj2.contains("b"));
System.out.println (conj.toString());
System.out.println (conj);
itens = conj.toArray();

for (int i =0; i< itens.length; i++)
    System.out.println ("vet["+i+"] = "+itens[i]);

for (String s : conj2)
    System.out.println (s);
    conj2.addAll(conj);
    System.out.println (conj2.toString());
for (String s : conj2)
    System.out.println (s);
}
```

É possível ter, dentre outros métodos, os descritos abaixo:

addAll ⇒ Adiciona todos os elementos de uma coleção em um conjunto

remove ⇒ Remove um objeto do conjunto

removeAll ⇒ Remove todos os elementos de uma coleção do conjunto

clear ⇒ Remove todos os elementos de um conjunto

isEmpty ⇒ Indica se um conjunto está vazio ou não

size ⇒ Indica quantos elementos pertencem ao conjunto

III.3 Verificação de Tipos

É a atividade de assegurar que os operandos de um operador sejam de tipos compatíveis.

Exemplo: não se pode somar o conteúdo de uma variável inteira com uma string.

Considere alguma operação que espera um operando (variável) do tipo T, mas que recebe um operando do tipo T'. Deve-se então checar se T é equivalente (compatível) a T'.

Em diversas situações, devemos decidir se dois objetos são do mesmo tipo, ou se são de tipos equivalentes, para que possam ser usados em uma determinada construção de uma linguagem. A forma da resposta depende, é claro, da linguagem, que deve definir suas próprias regras de equivalência de tipos.

A **Equivalência de Tipos** ou **Compatibilidade de Tipos** pode ser verificada de duas maneiras:

- **Equivalência estrutural**: $T \Rightarrow T'$ se e somente se T e T' possuem o mesmo conjunto de valores. Verifica-se o tipo declarado.
- **Equivalência nominal**: $T \Rightarrow T'$ se e somente se T e T' forem definidas no mesmo local ou em declarações que usem o mesmo nome do tipo.

Assim, podemos perceber que toda equivalência nominal também é estrutural, mas o contrário nem sempre é verdadeiro.

Exemplo:

```
TYPE
  T1 : ARRAY [ 1.. 10] OF INTEGER;
  T2 : ARRAY [ 1.. 10] OF INTEGER;
VAR
  X1, Y1 : T1;
  X2 : T2;
  X3 : T1;
```

X1 e X2 só são equivalentes estruturalmente.

X1, Y1 e X3 são equivalentes estruturalmente e nominalmente.

A linguagem Algol-68 utiliza a equivalência estrutural, cuja implementação é considerada um pouco mais trabalhosa: se dois tipos não tem o mesmo nome, é necessário verificar suas estruturas, num teste que pode envolver a verificação da equivalência de vários outros tipos, componentes dos tipos cuja equivalência se quer determinar.

Por outro lado, Ada adota a equivalência nominal pois considera de tipos distintos até mesmo duas variáveis introduzidas na mesma declaração: a declaração

x,y:array(1..10) of integer;

faz com que as variáveis x e y tenham tipos anônimos, que, *portanto*, não tem o mesmo nome.

E PASCAL ? mistura as duas equivalências. Estrutural para tipos simples e nominal para tipos compostos.

Exemplo:

```
TYPE
  T1 = integer;
  T2 = integer;
  T_V1 = ARRAY[1..5] OF INTEGER;
  T_V2 = ARRAY[1..6] OF INTEGER;
VAR
  N1 : T1;
  N2 : T2;
  V1, V3 : T_V1;
  V2: T_V2;
BEGIN
  N1 := N2; {aceita}
  V1 := V2; {não aceita}
  V1 := V3: {aceita}
END.
```

C usa equivalência estrutural para quase tudo exceto para registros e uniões.

C++ e Java usa equivalência de nomes.

A checagem de tipos feita pela linguagem pode ser:

- **Em tempo de compilação:** LP's são chamadas de estaticamente tipadas. Todo parâmetro e variável possuem um tipo fixo escolhido pelo programador. Deste

modo, o tipo de cada expressão pode ser deduzido e cada operação checada em tempo de compilação. A maior parte das LP's de alto nível são estaticamente tipadas. Menos custoso. Menos flexibilidade.

- **Em tempo de execução:** *LP's são chamadas de dinamicamente tipadas*. Somente os valores têm tipo fixo. Uma variável ou parâmetro não possuem tipo associado, mas podem designar valores de diferentes tipos em pontos distintos da execução. Isto implica que os tipos dos operandos devem ser checados imediatamente antes da execução da operação. Lisp, Apl e Smalltalk são LP's dinamicamente tipadas. Perda de eficiência. Mais flexibilidade.

Diz-se que uma linguagem é **fortemente tipada**, se é possível, apenas por análise *estática* do programa (análise realizada em tempo de compilação), determinar que a estrutura de tipos da linguagem não é violada. Uma linguagem que não garante a ausência de erros de tipos, pelo menos de forma estática, é dita **fracamente tipada**.

Um tipo pode ser violado de forma estática (atribuição) ou dinâmica (leitura de dado).

Ex.:

```
PROGRAM TESTE;  
TYPE  
    NATURAL=1..32767;  
VAR  
    I: INTEGER;  
    S: SHORTINT;  
    N: NATURAL;  
BEGIN  
    I:=10;  
    S:=-10;  
    N:=I;  
    N:=S; {VIOLAÇÃO DINÂMICA DE TIPO}  
    WRITELN(N); {mostra -10 na tela}  
    N:= -10; {VIOLAÇÃO ESTÁTICA DE TIPO}  
    I:=32767;  
    WRITELN(I); {mostra 32767 na tela}  
    S:=I; {VIOLAÇÃO DINÂMICA DE TIPO}  
    WRITELN(S); {mostra -1 na tela}  
END.
```

Exemplos de linguagens fortemente tipadas são Algol-68 e Ada; Pascal seria fortemente tipada com algumas pequenas modificações, tais como a definição de tipos procedimento, e a definição de sub-faixa.

C, C++ também não é fortemente tipado

ML é fortemente tipada.

Se, em alguma situação, é necessário um valor de um tipo **t1**, e apenas está disponível um valor de um tipo **t2**, pode ser necessária uma conversão de tipo.

Assim, quando os tipos envolvidos em uma expressão não são equivalentes, a linguagem pode oferecer a possibilidade de conversão de tipos. Um tipo compatível é aquele válido para o operador ou com permissão nas regras da linguagem para ser convertido pelo código gerado pelo compilador para um tipo válido. Existe dois tipos de conversão:

A **conversão implícita (automática)**, chamada de **coerção**, é feita com base nas regras definidas, sem interferência do programador. Desta forma, o código de conversão, (por exemplo do valor inteiro de *i* para o real equivalente) quando necessário, deve ser incluído pelo compilador. Esta forma de conversão é às vezes conhecida como coerção. Por exemplo, a maioria das linguagens aceita um comando `x := i;` mesmo quando *x* é uma variável real, e *i* uma variável inteira.

A **conversão explícita** ocorre quando há várias maneiras de converter um valor de um tipo **t1** para o tipo **t2**, é necessário indicar a forma de conversão desejada, e, em geral, isso é feito através de uma função de conversão. Este é o caso da conversão de reais em inteiros, que pode ser feita por arredondamento, truncamento, etc. Normalmente, funções pré-definidas da linguagem estão disponíveis para as situações mais comuns, incluindo casos como arredondamento, truncamento, aproximação superior (ceiling) ou inferior (floor). É realizada diretamente pelo programador e é chamada de **casting**.

A conversão de tipos primitivos é ditada pelas regras da linguagem. Normalmente, é admitida uma conversão de um tipo de representação menor para um de representação maior.

Vejamos como exemplo o Java:

De	Para
byte	short, int, long, float, double
short	int, long, float, double
char	int, long, float, double
int	long, float, double
long	float, double
float	double

A conversão pode ser feita de três maneiras:

- ✎ Por atribuição através de coerção para o tipo do lado esquerdo da atribuição
- ✎ Por promoção aritmética para o tipo de resultado esperado da operação (inferência)
- ✎ Por conversão explícita (casting)

Exemplo:

```
float f1, f2;  
int i;  
f1 = i; // atribuição  
f2 = f1/i; // promoção  
i = (int) f; // casting
```

A coerção contribui para que uma linguagem seja fortemente tipada ?

III.4 Lista de Exercícios

- 1) Defina valor de variável em uma LP e tipo.
- 2) Defina tipo primitivo e tipo composto.

- 3) Explique o conceito de cardinalidade de um tipo.
- 4) O que é um descritor?
- 5) Explique a diferença entre o tipo ponto flutuante e decimal. Quando se deve usar um e quando se deve usar o outro?
- 6) O que é uma união disjunta? Qual a vantagem de uma união disjunta em relação a um registro? Qual o problema que esse tipo pode acarretar?
- 7) Defina o que são tipos *enumerados* e *sub-faixa*.
- 8) Defina o que são vetores Estáticos, Semi-estáticos, Semi-dinâmicos e Dinâmicos. Quais as vantagens de cada tipo?
- 9) Que recurso de inicialização de vetores está presente em ADA, mas não em outras linguagens imperativas comuns?
- 10) Defina referências amplamente qualificadas e referências elípticas a campos em registros.
- 11) Quais os problemas que podem ocorrer com ponteiros?
- 12) Qual a vantagem de se tratar strings como um tipo primitivo?
- 13) Atualmente a tabela ASCII atende a todas as linguagens de programação? Justifique.
- 14) O que é coerção? Toda LP possui coerção?
- 15) Defina *erro de tipo*, *verificação de tipos* e *tipificação forte*.
- 16) Uma LP que realiza a verificação de tipos dinamicamente é mais eficiente do que uma LP que realiza a verificação de tipos estaticamente? Explique.
- 17) Analise o código abaixo. Se a equivalência dos tipos for estrutural este programa compilará sem erros? E se for nominal? Explique.

```
PROGRAM EX;
TYPE
  Tp_vet = array[1..100] of INTEGER;
  Tp_vet2 = array[1..100] of INTEGER;
  Inteiro = integer;
VAR
  A : INTEGER;
  v1: Tp_vet;
  B : Inteiro;
PROCEDURE TESTE (v:Tp_vet2);
  BEGIN
    :
  END;
BEGIN
  A := 11;
  B := A;
  TESTE (v1);
END.
```


CAPÍTULO IV - EXPRESSÕES E ATRIBUIÇÃO

Expressões são o meio fundamental de especificar computação em uma linguagem de programação.

Elas são compostas de operadores e operandos, devem ser avaliadas e produzem algum resultado. Os operadores indicam a operação que deverá ser realizada. Esta operação necessita ser validada, ou seja, deve-se verificar se a operação é compatível com o tipo de operando passado e com a quantidade de operandos envolvidos.

Para saber qual o resultado que uma expressão dará em uma determinada linguagem, é necessário saber em que ordem esta linguagem realiza a avaliação de operadores e operandos, ou seja, suas regras de associatividade e precedência.

IV.1 Expressões aritméticas

Nas LP's, as expressões aritméticas consistem em operadores, operandos, parênteses e chamadas a funções cuja finalidade é especificar uma computação aritmética.

Pode-se ter operadores:

- **Unário** ⇒ Possui um único operando: (-)
- **Binário** ⇒ Possui dois operandos: (+, -, /, *, etc.)
- **Ternário** ⇒ Possui três operandos (o operador ? do C, C++ e Java)

Notação

Na maioria das linguagens imperativas os operadores binários são **infixos**. No LISP, eles são **prefixos**.

IV.1.1 Ordem de avaliação de operadores

É a ordem em que os operadores são executados.

Precedência

Define a ordem em que operadores de diferentes níveis de prioridade são executados. A maioria das linguagens obedece à ordem de precedência matemática.

Exemplo:

	Pascal	C++, Java
Mais alta	*, /, div, mod +, - todos	++, -- (pré) +, - unário *, /, % +, - binário
Mais baixa		++, -- (pós)

Associatividade

Quando uma expressão contém mais de um operador do mesmo nível de precedência, a ordem em que eles serão executados depende das regras de associatividade da linguagem.

Na maioria das linguagens a regra de associatividade define que a execução ocorrerá da esquerda para a direita, exceto com o operador de exponenciação, se houver na linguagem, cuja execução ocorrerá da direita para a esquerda.

Exemplo:

- **FORTRAN** $\Rightarrow A ** B ** C$ (O operador direita é avaliado primeiro)
- **ADA** $\Rightarrow A ** B ** C$ (é ilegal pois não é associativa. O programador deve especificar, através de parênteses, qual operação deve ser realizada primeiro).

Em APL todos os operadores possuem a mesma ordem de precedência, a regra de associatividade é da direita para a esquerda.

Parênteses

Podem ser usados para alterar as regras de precedência e associatividade das expressões. O seu uso aumenta a legibilidade, pois o programador não precisa lembrar de regras.

Ex. $(a + b) * c$

Expressões condicionais

Em C, C++ e JAVA, existe um operador ternário **?** que é usado para formar expressões condicionais que têm somente uma linha de comando. Sua sintaxe é:

expressão_1 ? expressão_2 : expressão_3

Podemos ver no exemplo abaixo:

Condicional usando if-then-else	Condicional usando ?
<pre>if (cont == 0) media = 0; else media = soma / cont;</pre>	<pre>media = (cont == 0) ? 0 : soma / cont;</pre>

Assim, **?** denota o início da cláusula **then** e **:**, o início do **else**. Ambas são obrigatórias.

IV.1.2 Ordem de avaliação dos operandos

Será que esta ordem interfere em alguma coisa? Só se houver efeitos colaterais.

Efeitos Colaterais Funcionais \Rightarrow ocorre quando uma função modifica um parâmetro ou uma variável global.

Exemplo:

```
A := 10;
x := A + func(A);
```

onde,

```
function func (var A: integer): integer;
begin
  A := 5;
  func := 10;
end;
```

Neste caso, a ordem da avaliação produz efeito sobre o resultado, pois:

- Se o valor de A for buscado primeiro, seu valor será 10 e a expressão valerá 20.
- Se o segundo operando for avaliado primeiro, A terá valor 5 e a expressão valerá 15.

Veja no exemplo abaixo:

```
int a = 5
int funcao1 ( ) {
    a = 17;
    return 3;
} /* fim da funcao1 */

void funcao2 ( ) {
    a = a + funcao1( );
} /* fim da funcao2 */

void main ( ) {
    funcao2( );
} /* fim do main */
```

O valor de **a** em `funcao2()` depende da ordem de avaliação, definida pela linguagem, sobre os operandos na expressão **a = a + funcao1 ()**; Assim, **a** poderá ser 8 ou 20.

Solução:

- evitar variáveis globais dentro de funções ou procedimentos
- ver a ordem de avaliação da LP. Há, por exemplo, LP's que avaliam da esquerda para a direita (JAVA), outras que permitem que o implementador decida a ordem (ADA)

Operadores sobrecarregados

Um operador possui mais de uma finalidade. Por exemplo:

- no Pascal, o operador **+** pode ser usado para adição de tipos numéricos, mas pode também ser usado para concatenar strings. Mas, esse é um exemplo em que a sobrecarga não prejudica a legibilidade.
- No C, o operador **&** pode ser um operador binário (significando AND) ou unário (significando o endereço de uma variável). Neste caso, como as operações realizadas não têm nenhuma relação entre si, a sobrecarga prejudica a legibilidade.
- No C++ e no Java, o operador **/** indica a divisão entre dois números. Se os dois forem inteiros, o resultado será um inteiro (truncado). Isso pode provocar um erro de lógica no programa, pois se tiver um comando **media = total/quant** onde **media** é um real e **total(250)** e **quant(100)** inteiros, o resultado dará um inteiro truncado (**2**) que será convertido para real. Portanto, a média teria como resultado 2 e não 2.5 como deveria ser. No Pascal isso não acontece pois o operador que realiza a divisão por inteiros é **div**.

Algumas linguagens que suportam tipos abstratos de dado (C++, ADA, etc) permitem que o programador sobrecarregue mais ainda seus operadores. Por exemplo, é possível definir um operador **+** para realizar uma adição de vetores. Mas isso não impediria que o programador definisse **+** para significar multiplicação, por exemplo.

Conversão de tipo

- Conversão de Estreitamento \Rightarrow converte um tipo em outro de faixa inferior. Exemplo, converter um **longint** para um **int**, cuja faixa de valores é menor
- Conversão de Alargamento \Rightarrow converte um tipo em outro de faixa maior. Exemplo, converter um **int** para um **real**, cuja faixa de valores é maior.
- Expressões de modo misto \Rightarrow um operador permite operandos de tipos diferentes.

Existem dois tipos de conversões de valores:

✎ **Coerção** \Rightarrow Conversão de tipos implícita.

Exemplo: JAVA

```
void meumetodo ( ){  
    int a, b, c;  
    char d;  
    ...  
    a = b * d;  
    ...  
}
```

O JAVA aceita um inteiro receber um caractere, pois o caractere será automaticamente convertido para o seu código na tabela UNICODE. Mas isso também pode gerar um problema. Se houve um erro e, ao invés de d, deveria ser digitado c?

Obs.: O C++ e o JAVA têm tipos inteiros menores do que o **int** (**char** e **short int** – C++ e **byte**, **short** e **char** – JAVA) que, quando um operador é aplicado sobre eles, eles são convertidos para **int** e, depois retornados a seus valores originais.

✎ **Cast** \Rightarrow Conversão explícita de tipos.

Exemplo: JAVA

```
void meumetodo ( ){  
    int a, b, c;  
    double d;  
    ...  
    a = b * (int)d;  
    ...  
}
```

Erros em Expressões

Limitações da aritmética do computador:

Overflow \Rightarrow O resultado é muito grande não podendo ser armazenado na célula de memória.

Underflow \Rightarrow o resultado é muito pequeno não podendo ser representado na célula de memória

Limitações inerentes à aritmética \Rightarrow Divisão por zero

IV.2 Expressões Relacionais

Tem 2 operandos e um operador relacional : >, <, <=, >=, < >, =

Os operadores relacionais possuem menor precedência do que os aritméticos

a + 1 > 2 * 2 (as operações aritméticas são realizadas primeiro)

IV.3 Expressões Booleanas

- Consistem em variáveis, constantes, expressões relacionais e operadores booleanos: AND, OR, NOT, às vezes, OR exclusivo e um para equivalência.
- 2 operandos e um operador
- 1 operando e um operador

No PASCAL os operadores booleanos têm maior precedência que os relacionais. Portanto, **a > 5 or a < 0** não funciona (pois 5 não é operando booleano). O correto seria **(a > 5) or (a < 0)**

IV.4 Avaliação Curto-Circuito

Uma avaliação curto-circuito de uma expressão tem seu resultado determinado sem avaliar todos os operandos e ou operadores. Ex:

(13*A)*(B/13-1) se A = 0 não precisa de avaliar o resto, pois o resultado será 0.

(A>=0) and (B<0) se o 1º for Falso não precisa de avaliar o resto

C possui avaliação curto-circuito: **(a>b) || (b++/3)**

Assim, b só será incrementado se a <= b (efeito colateral)

Observemos o código abaixo em Java. Supondo uma variável do tipo VETOR que possui TAM elementos, que será pesquisada a procura de um valor CHAVE:

```
ind = 1;
while (ind<vetor.length && vetor[ind] != chave) do
    ind = ind+1;
```

Se a avaliação não for feita com curto-circuito, ambas as expressões relacionais serão avaliadas, independente do valor da primeira. Isto poderia acarretar um erro de subscrito, se CHAVE não estivesse no VETOR, pois cairia fora da extensão do vetor.

IV.5 Atribuição

É o mecanismo pelo qual a variável é modificada dinamicamente.

➤ Atribuições simples

Sintaxe: <nome_variável> <comando atribuição> <expressão>

- **Alvos múltiplos** ⇒ permite a atribuição de um valor a mais de uma variável.

Exemplo: soma, total = 0;

C, C++

- **Alvos condicionais** ⇒ Dependendo de uma condição o valor é atribuído a uma variável ou a outra.

Exemplo: `flag ? cont1 : cont2 = 0;`

Isto é equivalente a: `IF (flag) THEN cont1 := 0
ELSE cont2 := 0;`

- **Operadores de atribuição compostos** ⇒ São comandos de atribuição que agregam, além da função de atribuir, uma outra representada simbolicamente.

Exemplo em Java: `soma += valor;`

Equivalente a: `soma = soma + valor;`

O C, C++ e o Java têm versões de operadores de atribuição compostos para a maioria de seus operadores binários.

- **Operadores de atribuição unários** ⇒ São comandos de atribuição que agregam, além da função de atribuir, uma outra mas não são representados por 2 símbolos.

Com Operadores Unários	Sem Operadores Unários
<code>soma = ++cont;</code>	<code>cont = cont + 1 soma = cont;</code>
<code>soma = cont++;</code>	<code>soma = cont; cont = cont + 1;</code>
<code>soma = -cont++;</code>	<code>soma = - cont; cont = cont + 1;</code>
<code>soma = - ++cont;</code>	<code>cont = cont + 1; soma = - cont;</code>

- **Atribuição em Expressões** ⇒ Quando um comando de atribuição é utilizado no meio de uma expressão. LP's como C, C++ e JAVA permitem esse tipo de atribuição.

`while ((ch = getchar()) != EOF) {....}`

Esse tipo de atribuição pode causar expressões difíceis de ler e entender, como por exemplo:

`A = B+(C=D/B++)-1;`

(a ordem seria: `C = D / B; B = B + 1; A = B(antigo) + C -1`)

Também é possível, no C, C++ e Java realizar atribuições como:

`soma = cont = 0;`

Como C e C++ aceitam expressões numéricas em suas instruções if, isso pode aumentar a probabilidade de erros de programas, pois ao digitar

`if (x = y) . . .`

ao invés de

`if (x == y) . . .`

O compilador não detectará este erro. JAVA só permite expressões booleanas em suas condicionais.

Exemplo de uma programa em Java com coerção, casting, pós e pré incremento e atribuição em expressão:

```
public class Coerção {
    public static void main(String[] args){
        int num;
        double a, b,c;
        char car = 'a';
        float d = 3.5f;
        a = 1.0;
        b = 2.0;
        num = (int)(a*b); //num = 2
        num = num*(int)d; // num = 2*3 = 6
        num = (int)(num*d); // num = 6*3.5 = 21
        num = car*2;//num = 194
        a = b = 0;//a = 0.0    e    b = 0.0
        num = 3;
        b = 1;
        a = - num + b;//a = -3+1 = -2
        a = num + -b++;//a = 3+-1.0 = 2.0    e    b = 2.0
        a = num + -++b;//b = 3.0    e    a = 3+(-3.0) = 0.0
        a = -b++;//a = -3.0    e    b = 4.0;
        a = -++b;//b = 5.0    e    a = -5.0
        if ((a=b)!=0);
        a = b+(c=d/b++)-1;//a = 4.7    b = 6.0    e    c = 0.7
        System.out.println("RESULTADO\ na = "+a+"\nb = "+b+
                           "\nnum = "+num);

        b = 5;
        a = (c=d/b++)+b-1;// a = 5.7    b = 6.0    e    c = 0.7
        System.out.println("RESULTADO\ na = "+a+"\nb = "+b+
                           "\nnum = "+num);
    }
}
```

IV.6 Lista de Exercícios

- 1) O que é precedência de operadores e associatividade de operadores?
- 2) O que é um operador sobrecarregado?
- 3) Defina efeito colateral funcional.
- 4) Defina conversões de estreitamento e de alargamento.
- 5) O que é uma expressão de modo-misto?
- 6) Para que serve um operador de atribuição composto? Dê um exemplo.
- 7) Em algumas linguagens, pode-se tratar um operador de atribuição como operador aritmético. Quais as desvantagens que isto pode acarretar?
- 8) Qual a precedência dos operadores unários em Java?

9) Você acha que eliminar os operadores sobrecarregados de uma linguagem seria bom? Por quê?

10) Seria bom eliminar todas as regras de precedência de operadores exigindo, assim, parênteses em todas as expressões? Por que?

11) Descreva uma situação em que o operador de adição em uma LP não seria comutativo.

12) Mostre a ordem de avaliação das seguintes expressões como o exemplo abaixo:

$$a + b * c + d \Rightarrow (a + (b * c)^1)^2 + d^3$$

a) $a * b - c + d$

b) $a * (b - c) / d \bmod 10$

c) $(a - b) / c \text{ and } (d * e / a - 3)$

d) $-a + b$

e) $-a \text{ or } c = d \text{ and } b$

13) Mostre a ordem de precedência do exercício anterior, supondo que a linguagem não tenha regra de precedência e que os operadores associem-se da direita para a esquerda.

14) Supondo o programa abaixo:

```
function F1 (var tam : integer) : integer;
```

```
begin
```

```
    tam := tam + 1;
```

```
    F1 := 2 * tam - 5;
```

```
end;
```

```
{programa principal}
```

```
begin
```

```
    i := 1;
```

```
    soma := (2 * i) + f1 ( i );
```

```
    j := 1;
```

```
    soma2 :=- f1 ( j ) + (2 * j);
```

```
end.
```

a) Supondo que os operandos são avaliados da esquerda para direita, quais os valores de soma e soma2?

b) Supondo que os operandos são avaliados da direita para esquerda, quais os valores de soma e soma2?

CAPÍTULO V – VARIÁVEIS

“Uma vez que o programador tenha entendido o uso de variáveis, ele entendeu a essência da programação”. Dijkstra

No paradigma imperativo:

- Uma variável é um objeto que contém um valor, o qual pode ser inspecionado e atualizado sempre que necessário. São usadas para modelar objetos do mundo real que possuem estados.
- É uma abstração de uma célula ou de um conjunto de células de memória de um computador. Ou seja, é a substituição de um endereço numérico de memória por um nome, tornando os programas mais legíveis, mais fáceis de escrever e de manter.

Uma variável pode ser caracterizada como um sêxtuplo de atributos: **nome, endereço, valor, tipo, tempo de vida e escopo.**

V.1 Nomes

O nome é uma cadeia de caractere usada para identificar uma entidade do programa. Os nomes, ou identificadores, obedecem a uma regra de formação de identificadores da LP. Algumas linguagens fazem distinção entre maiúscula e minúscula (Isto pode ser uma vantagem ou uma desvantagem).

PRIMEIRAS LINGUAGENS = nomes de 1 caractere (matemáticos)

FORTRAN I = nomes de até 6 caracteres

FORTRAN 90 e C = 31 caracteres

ADA = não tem limite de tamanho

Palavras especiais

a) **palavras reservadas** = não podem ser usadas para nenhum outro fim

b) **palavra-chave** = são especiais apenas em certos contextos, por exemplo:

em FORTRAN: (usa nos dois sentidos)

REAL N {Tipo}

REAL = 3.4 {nome de uma variável}

Problemas de legibilidade.

c) **nomes predefinidos** = possuem um significado predefinido, mas podem ser redefinidos. Exemplo: readln e o writeln em PASCAL. (Define um outro sentido, não pode ser mais usado com o anterior).

```
VAR
  readln : INTEGER;
BEGIN
  readln := 1;
END.
```

Obs: Existem variáveis que não possuem nome: os nós alocados dinamicamente.

V.2 Endereço

É o endereço de memória onde o espaço para a variável foi alocado, ou seja, é a posição na memória da primeira célula ocupada pela variável. Mas, em muitas linguagens, é possível ter o mesmo nome associado a lugares diferentes da memória, em tempos diferentes no programa.

Então, é possível ter:

- ✍ Variáveis com mesmo nome acessando lugares diferentes da memória, em tempos diferentes.
- ✍ Variáveis com nomes diferentes acessando o mesmo endereço de memória (esses nomes são chamados apelidos ou alíases).

V.3 Tipo

Determina a faixa de valores que a variável pode assumir. O tipo pode ser declarado **explicitamente**, através de declaração de variáveis com o tipo definido, ou **implicitamente**, onde a primeira ocorrência do nome da variável no programa constitui sua declaração.

A maioria das linguagens de programação exige declarações explícitas, pois as implícitas podem prejudicar a legibilidade e porque podem impedir que o compilador detecte erros tipográficos e do programador.

Existem LP's, como o JavaScript, em que os tipos são vinculados dinamicamente, ou seja, não há especificação do tipo da variável na sua declaração. À medida que a variável recebe um determinado valor, ela é vinculada ao tipo do valor.

Ex.:

```
LIST = [10.2 5.1 0.0]  
LIST = 47
```

A primeira instrução faz com que ela se torne um array unidimensional de tamanho 3 e de elementos do tipo real. A segunda instrução faz com que a variável torne-se do tipo inteiro.

Desvantagens:

- ✍ Diminui a capacidade de detecção de erros do compilador, pois dois tipos quaisquer podem aparecer em lados opostos de uma operação de atribuição
- ✍ O custo para implementar este tipo de vinculação é alto, principalmente durante a execução, pois a verificação de tipos ocorrerá em tempo de execução, a variável deverá ter um descritor que armazenará seu tipo corrente e o espaço de memória reservado para uma variável deve ter tamanho variável já que o tipo da variável também pode variar.

Linguagens que usam vinculação dinâmica, geralmente, são linguagens interpretadas.

V.4 Valor

É o conteúdo da célula ou das células de memória associadas à variável e deve ser compatível com seu tipo.

V.5 Tempo de vida

Uma variável pode ser criada (alocada) e em um outro instante eliminada (desalocada). O intervalo de tempo existente entre a criação e a destruição de uma variável é chamado de **tempo de vida** da variável.

De acordo com o tempo de vida podemos classificar quatro categorias de variáveis escalares (não estruturadas):

- **Variáveis estáticas**

São aquelas alocadas antes que a execução do programa inicie e assim permanecem até que a execução do programa encerre.

São as variáveis globais.

Vantagens:

- ✎ Eficiência (endereçamento direto e não gasta tempo de execução do programa para alocar e/ou desalocar espaço em memória).
- ✎ Sensíveis a história

Desvantagem:

- ✎ Falta de flexibilidade (Uma linguagem que só possui variáveis estáticas não permite recursividade).
- ✎ Armazenamento não pode ser compartilhado

- **Variáveis stack-dinâmicas**

São aquelas alocadas na pilha em tempo de execução, ao chamar um subprograma, por exemplo. Elas são declaradas na codificação do programa e seus tipos são estaticamente verificados. Essas variáveis são alocadas na pilha em tempo de execução.

Vantagens:

- ✎ Economia de memória
- ✎ Flexibilidade (Na recursão, é necessário manter uma pilha de variáveis locais para cada chamada recursiva).

Desvantagem:

- ✎ Perda de tempo para a alocação e desalocação de memória.

Em linguagens como Pascal, Java e C++, as variáveis locais são assumidas como dinâmicas na pilha.

- **Variáveis heap-dinâmicas explícitas**

São células de memória sem nome (anônimas), alocadas na heap em tempo de execução, que podem ser acessadas através de ponteiros.

Geralmente utilizam de um comando para a alocação e outro para a desalocação. Vejamos alguns exemplos de linguagens com esses tipos de comandos:

Pascal

```
type
    nodo = record
        dado: integer;
        prox : ^nodo;
    end;
var
    elem : ^nodo;
begin
    ...
    new (elem); {Aloca um espaço na memória para um tipo inteiro
                (dado) e um tipo ponteiro (prox)}
    ...
    dispose (elem); {desaloca a célula para a qual elem aponta}
    ...
end;
```

C++

```
int *elem; // declaração não aloca espaço
...
elem = new int; //Aloca um espaço na memória para um tipo inteiro
...
delete elem; //desaloca a célula para a qual nodo aponta
...
```

Em ambas as linguagens, uma variável é criada pelo operador **new**. Esta variável não tem nome e deverá ser referenciada pelo ponteiro **elem**. Depois, a variável é desalocada pelo operador **dispose** (Pascal) ou **delete** (C++).

Em Java, todos os dados que não são escalares primitivos são objetos. Os objetos são dinâmicos e, portanto, acessados através de ponteiros. A alocação do objeto no Java segue o padrão do C++ com o comando **new**, mas não se usa nenhum comando de desalocação da variável por causa do coletor de lixo.

Variáveis dinâmicas explícitas são usadas, na maioria das vezes, para implementar estruturas dinâmicas (listas, pilhas, filas, árvores, etc).

Vantagem

- ✎ Economia de memória

Desvantagem:

- ✎ Perda de tempo para a alocação
- ✎ Dificuldades de usar corretamente

- **Variáveis heap-dinâmicas implícitas**

São aquelas que não precisam de declaração prévia. Armazenam qualquer tipo e o seu tipo é atribuído quando lhe são atribuídos valores.

Vantagem

- ✎ Flexibilidade (código altamente genérico)

Desvantagem

- ✎ Sobretaxa de manter todos os atributos dinâmicos em tempo de execução.
- ✎ Perda de certa capacidade de detecção de erros pelo compilador.
- ✎ Perda de tempo para gerenciar estas variáveis.

Exemplos:

Algol 68, APL

V.6 Escopo

É a faixa de instruções na qual a variável é visível, ou seja, na qual ela pode ser referenciada.

As regras de escopo de um LP determinam como uma variável declarada fora de um subprograma ou bloco pode ser ali acessada.

Existem dois tipos de escopo:

a) o Estático

Quando o escopo de uma variável pode ser determinado antes da execução de um programa.

Em algumas linguagens como Pascal, Ada e Java-Script, os sub-programas podem ser aninhados dentro de outros sub-programas, criando assim, uma hierarquia de escopos em um programa.

Exemplo:

<pre>PROGRAM EXEMPLO; VAR X : INTEGER; PROCEDURE TESTE1; VAR X : INTEGER; BEGIN WRITELN (X); END;</pre>	<pre>PROCEDURE TESTE2; BEGIN WRITELN (X); END; BEGIN ... END.</pre>
--	---

A variável x global ficou oculta pelo variável x local de Teste1. Em geral, uma declaração para uma variável oculta qualquer declaração de outra variável com o mesmo nome em um escopo envolvente maior.

Algumas LP's permitem que sejam acessadas as variáveis ocultas a um procedimento. Por exemplo, em C++, as globais ocultas podem ser acessadas assim: `::x`

Algumas linguagens como C, C++ e Java permitem que qualquer instrução composta tenha declarações e, assim, defina um novo escopo. Por exemplo, suponha **list** um vetor de inteiros:

```
if (list[i] < list[j]) {  
  int temp;  
  temp = list[i];  
  list[i] = list[j];  
  list[j] = temp;  
}
```

Os escopos criados por blocos são tratados da mesma forma que aqueles criados por subprogramas, ou seja, seu escopo é somente dentro do bloco em que foi declarada.

Variáveis que são referenciadas em um bloco, mas não são declaradas dentro do mesmo são conectadas a declarações em escopos maiores.

Linguagens como C++ e Java permitem que variáveis sejam definidas em qualquer lugar de uma função. Mas seu escopo será da posição da declaração até o fim da função.

Uma instrução **for** do C++ e do Java permitem declarações de variáveis em sua inicialização e o escopo dessa variável se restringe, na versão padrão do C++ e no Java, à construção do **for**.

Em linguagens como o Pascal, em que é possível declarar sub-rotinas dentro de sub-rotinas, as variáveis visíveis na sub-rotina mais interna são aquelas declaradas na própria sub-rotina mais as declaradas na mais externa mais as declaradas no programa principal.

```

PROGRAM EXEMPLO;
VAR X : INTEGER;
PROCEDURE TESTE3;
  VAR Y : INTEGER;
  BEGIN
    ... ← T3
  END;
PROCEDURE TESTE1;
  VAR A : INTEGER;
  PROCEDURE TESTE2;
    VAR B : INTEGER;
    BEGIN {TESTE2}
      ... ← T2
    END; {TESTE2}
  BEGIN {TESTE1}
    ... ← T1
  END; {TESTE1}
BEGIN {EXEMPLO}
  ... ← EX
END. {EXEMPLO}

```

Podemos observar no exemplo acima que o escopo de cada uma das variáveis será:

VARIÁVEL	DECLARAÇÃO	ESCOPO
<i>X</i>	<i>Programa Principal</i>	<i>Todo o programa</i>
<i>Y</i>	<i>Teste3</i>	<i>Teste3</i>
<i>A</i>	<i>Teste1</i>	<i>Teste1 e Teste2</i>
<i>B</i>	<i>Teste2</i>	<i>Teste2</i>

Mas, em todos os casos, a verificação da declaração da variável ocorre do bloco mais interno para o bloco mais externo.

b) o Dinâmico

Baseia-se na sequência de chamadas de subprogramas, não em suas relações espaciais um com o outro. Assim, o escopo será determinado em tempo de execução.

```
PROGRAM TESTE;  
USES CRT;  
VAR X: INTEGER;  
  
PROCEDURE TESTE1;  
BEGIN  
    WRITELN (X);  
END;  
  
PROCEDURE TESTE2;  
VAR X : INTEGER;  
BEGIN  
    X := 15;  
    TESTE1;  
END;  
  
BEGIN  
    X := 10;  
    TESTE1;  
    TESTE2;  
END.
```

O que será impresso no vídeo?

10

10

Pois o escopo do pascal é estático, ou seja teste1 só consegue ver X global e não o local.

Se fosse escopo dinâmico imprimiria

10

15

Características:

- ✎ Não precisa passar variáveis.
- ✎ A verificação das variáveis é feita em tempo de compilação; é mais complicado de realizar ⇒ pois é incapaz de, estaticamente, verificar o tipo das referências a variáveis não-locais. Portanto, o programa é incapaz de determinar estaticamente a declaração para uma variável referenciada como não-local.
- ✎ Este tipo de escopo resulta em programas menos confiáveis ⇒ porque quando um sub-programa inicia sua execução, suas variáveis locais são todas visíveis a qualquer outro sub-programa em execução.
- ✎ Pode dificultar a leitura do programa (deve-se saber a ordem de chamada). Assim, provoca **ilegibilidade** ⇒ porque a sequência de chamadas a sub-programas deve ser conhecida para determinar o significado das referências a variáveis não-locais.
- ✎ Perde tempo na execução para a verificação

Exemplos: APL, SNOBOL4 e as primeiras versões de LISP.

• Escopo x Tempo de vida

Tempo de vida é o tempo de existência da variável, ou seja, é o intervalo de tempo em que a variável está alocada na memória.

Escopo é referente ao local onde o acesso à variável é permitido.

Normalmente, o tempo de vida de uma variável local é o intervalo de tempo entre a chamada do subprograma em que ela foi declarada e o final de sua execução. Mas,

existem exceções, como as variáveis declaradas como **static** no C++, cujo escopo é local mas o tempo de vida é durante toda a execução do programa (Veja exemplo abaixo).

Exemplo:

Em C:

```
int teste (void)
{ static int num = 100;
  num = num + 1;
  return (num);
}
void main (void)
{ int i;
  i = teste;
  printf ( "%i ", i);
}
```

A variável existe durante toda a execução do programa, mas o seu escopo pertence somente a teste.

V.7 Constantes Nomeadas

É uma variável que armazena um valor fixo e este não pode ser alterado pelo programa. São importantes, pois aumentam a legibilidade e confiabilidade do programa, além de facilitar quando o programa utiliza um número de fixo de valores de dados e depois deseja alterá-lo.

As declarações de constantes nomeadas em Pascal devem ter valores simples do lado direito do operador =. Ada, C++ e Java permitem vinculação dinâmica de valores para constantes nomeadas. Então, expressões com variáveis podem ser atribuídas às constantes na declaração.

V.8 Inicialização de Variáveis

A inicialização de variáveis muitas vezes pode ocorrer no momento de sua declaração, quando a linguagem permitir, ou em tempo de execução.

V.9 Variáveis Transientes e Variáveis Persistentes

Variáveis Transientes: variáveis cujo tempo de vida é limitado pela ativação do programa que a criou. Ex. variáveis locais.

Variáveis Persistentes: são aquelas cujo tempo de vida transcende a ativação de um programa em particular Ex. arquivos.

V.10 Lista de Exercícios

- 1) Qual é o perigo potencial das LP's que fazem distinção entre letras maiúsculas e minúsculas?
- 2) De que maneira as palavras reservadas são melhores do que as palavras-chave?
- 3) O que é um apelido?
- 4) Defina variáveis estáticas, dinâmicas na pilha, dinâmicas explícitas e dinâmicas implícitas. Quais as vantagens e desvantagens de cada uma?
- 5) Defina tempo de vida, escopo.
- 6) Defina escopo estático e escopo dinâmico. Quais as vantagens e desvantagens de cada um?
- 7) Quais as vantagens das constantes nomeadas?

- 8) Dê um exemplo em que uma variável sensível à história é útil em um subprograma.
- 9) Qual o escopo e o tempo de vida de uma variável declarada como **static** no C++?
- 10) Considere o programa em Pascal esquemático abaixo:

```
PROGRAM EX;
  VAR X : INTEGER;

PROCEDURE SUB1;
  BEGIN { SUB1 }
    WRITELN ('X = ', X);
  END; { SUB1 };
```

```
PROCEDURE SUB2;
  VAR X : INTEGER;
  BEGIN { SUB2 }
    X := 10;
    SUB1;
  END; { SUB2 };

BEGIN { EX }
  X := 5;
  SUB2;
END. { EX };
```

- a) Suponha que este programa tenha sido compilado e executado usando as regras de escopo estático. Qual o valor de X que será mostrado na tela?
- b) E se as regras forem de escopo dinâmico, qual o valor de X que será mostrado na tela?

- 11) Considere o programa abaixo:

```
PROGRAM EX;
  VAR X, Y, Z : INTEGER;

PROCEDURE SUB1;
  VAR A, Y, Z : INTEGER;

  PROCEDURE SUB2;
    VAR A, B, Z : INTEGER;
    BEGIN { SUB2 }
      . . .
    END; { SUB2 };
  . . .
END; { SUB1 };
```

```
BEGIN { SUB1 }
  . . .
END; { SUB1 };

PROCEDURE SUB3;
  VAR A, X, W : INTEGER;
  BEGIN { SUB3 }
    . . .
  END; { SUB3 };

BEGIN { EX }
  . . .
END. { EX }
```

Liste todas as variáveis, juntamente com as unidades de programa em que elas foram declaradas que são visíveis nos corpos de SUB1, SUB2 e SUB3, supondo que seja usado o escopo estático.

- 12) Considere o programa abaixo:

```
PROGRAM EX;
  VAR X, Y, Z : INTEGER;

PROCEDURE SUB1;
  VAR A, Y, Z : INTEGER;
  BEGIN { SUB1 }
    . . .
  END; { SUB1 };

PROCEDURE SUB2;
  VAR A, X, W : INTEGER;
```

```
PROCEDURE SUB3;
  VAR A, B, Z : INTEGER;
  BEGIN { SUB3 }
    . . .
  END; { SUB3 };
  BEGIN { SUB2 }
    . . .
  END; { SUB2 };
BEGIN { EX }
  . . .
END. { EX }
```

Liste todas as variáveis, juntamente com as unidades de programa em que elas foram declaradas que são visíveis nos corpos de SUB1, SUB2 e SUB3, supondo que seja usado o escopo estático.

13) Considere o programa abaixo em C:

```
void fun (void) {
    int a, b, c; /* definição 1 */
    . . .
    while ( . . . ) {
        int b, c, d; /* definição 2 */
        . . . /* 1 */
        while ( . . . ) {
            int c, d, e; /* definição 3 */
            . . . /* 2 */
        }
        . . . /* 3 */
    }
    . . . /* 4 */
}
```

Para cada um dos quatro pontos marcados nessa função, liste cada variável visível, juntamente com o número da instrução de definição que a define.

14) Considere o programa abaixo:

<pre>PROGRAM EX; VAR X, Y, Z : INTEGER; PROCEDURE SUB1; VAR A, Y, Z : INTEGER; BEGIN { SUB1 } . . . END; { SUB1 }; PROCEDURE SUB2; VAR A, B, Z : INTEGER; BEGIN { SUB2 }</pre>	<pre> . . . END; { SUB2 }; PROCEDURE SUB3; VAR A, X, W : INTEGER; BEGIN { SUB3 } . . . END; { SUB3 }; BEGIN { EX } . . . END. { EX }</pre>
--	--

Dadas as seguintes sequências de chamada e supondo-se que seja usado o escopo dinâmico, quais variáveis são visíveis durante a execução do último subprograma ativado? Inclua, em cada variável visível, o nome da unidade em que ela foi declarada.

- EX chama SUB1; SUB1 chama SUB2; SUB2 chama SUB3.
- EX chama SUB1; SUB1 chama SUB3.
- EX chama SUB2; SUB2 chama SUB3; SUB3 chama SUB1.
- EX chama SUB3; SUB3 chama SUB1.
- EX chama SUB1; SUB1 chama SUB3; SUB3 chama SUB2.
- EX chama SUB3; SUB3 chama SUB2; SUB2 chama SUB1.

15) Com relação ao **procedimento P1** abaixo, responda aos itens (a), (b) e (c):

```
Program Exemplo;  
Var x : String;  
  
Procedure P1 (var x : String);  
Begin  
    ...  
End;
```

```
Var  
    a, b : String;  
Begin  
    ...  
    P1(b);  
    ...  
End;
```

- a) qual o escopo da variável x?
- b) qual o seu tempo de vida?
- c) Em que segmento fica armazenada?

CAPÍTULO VI - ABSTRAÇÃO DE CONTROLE

- **Instrução de Controle:** mecanismos que proporcionam selecionar entre caminhos alternativos do fluxo de controle e ou provocar a execução repetida de certos conjuntos de instruções.

Apenas duas instruções de controles seriam necessárias: uma condicional (**se**, por exemplo) e outra de iteração (**enquanto**, por exemplo).

- **Estrutura de Controle:** é uma instrução de controle e sua coleção de comandos cuja execução ela controla.

VI.1 – Instruções de seleção

Oferece meios de escolher entre dois ou mais caminhos.

VI.1.1 – Seletores Bidirecionais:

IF ... THEN ... ELSE

Neste tipo de instrução, podem ocorrer problemas como, por exemplo, em **if**'s aninhados. Veja o código abaixo em JAVA:

```
if (ok)
    if (total > 0)
        resultado = total/10;
    else
        resultado = 0;
```

Neste caso, embora pareça que o **else** seja relativo ao **if** mais externo, ele é relativo ao **if** mais interno, já que no JAVA, o **else** sempre faz par com o **if** mais recente.

Em linguagens como o ALGOL 60, não é permitido esse tipo de sintaxe, ou seja, quando se tem um **if** dentro de outro, é necessário que o segundo seja colocado em uma instrução composta de duas maneiras:

```
if (ok) then
begin
    if (total > 0) then
        resultado = total/10
    else
        resultado = 0
end
```

```
if (ok) then
begin
    if (total > 0) then
        resultado = total/10
    end
else
    resultado = 0
```

Esse tipo de controle também pode ser feito através de palavras especiais de fechamento de seleção, como em ADA:

```
if (OK) then
    if (TOTAL > 0) then
        RESULTADO = TOTAL/10;
    else
        RESULTADO = 0;
    end if;
end if;
```

```
if (OK) then
    if (TOTAL > 0) then
        RESULTADO = TOTAL/10;
    end if;
else
    RESULTADO = 0;
end if;
```

Assim, **else** sempre marca o final da cláusula **then**, mesmo se ela contiver várias instruções, e **end if**, por sua vez, marca o final de todo o **if**.

Dessa forma, mesmo que dentro do **if** ou do **else** se tenha mais de uma instrução, não há problemas com relação ao controle da instrução:

```
if A > B then
    TOTAL := TOTAL + A - B;
    CONT := CONT + A;
else
    TOTAL := TOTAL + B - A;
    CONT := CONT + B;
end if;
```

VI.1.2 – Seletores Múltiplos

Permitem a escolher uma opção dentre várias. Um seletor múltiplo pode ser construído também através do uso de **IF**'s aninhados mas, geralmente, o código fica confuso, difícil de escrever e ler e pouco confiável.

Exemplo em Pascal:

```
Case mes of
    2 : max := 28;
    4, 6, 9, 11 : max := 30;
    1, 3, 5, 7, 8, 10, 12 : max :=
31;
    else writeln ('ERRO');
end;
```

Exemplo em C, C++, Java:

```
switch (mes) {
    case 2: maxDia = 28; break;
    case 4:
    case 6:
    case 9:
    case 11: maxDia = 30; break;
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12: maxDia = 31; break;
    default: printf ("ERRO");
}
```

VI.2 – Instruções Iterativas

Faz com que uma ou mais instruções sejam executadas zero, uma ou mais vezes.

- ✓ *Corpo de um laço iterativo*: coleção de instruções controladas pelo comando de repetição.
- ✓ *Pré-teste de um laço*: quando o teste para finalização do laço ocorre antes do corpo do laço.
- ✓ *Pós-teste de um laço*: quando o teste para finalização do laço ocorre depois do corpo do laço.

VI.2.1 – Laços controlados por contador:

Exemplo: FOR. Possuem uma variável de laço onde os valores iniciais e finais são predeterminados e geralmente a diferença entre seus valores seqüenciais, stepsize, pode ser estabelecido.

Vejamos exemplos de algumas linguagens:

ALGOL 60 => há três formas mais simples e outras, mais complexas, combinando as primeiras:

1)

```
for cont := 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 do  
  Lista[cont] := 0
```

2)

```
for cont := 1 step 1 until 10 do  
  Lista[cont] := 0
```

3)

```
for cont := 1, cont + 1 while (cont <= 10) do  
  Lista[cont] := 0
```

4)

```
for indice := 1, 4, 13, 41  
  Step 2 until 47,  
  3 * indice while indice < 1000,  
  34, 2, -24 do  
  soma := soma + indice
```

Neste caso, **indice** terá os valores 1, 4, 13, 41, 43, 45, 47, 141, 423, 34, 2, -24

5)

```
i := 1;  
for cont := 1 step cont while (cont < 3*i) do  
  i := i + 1
```

Aqui nós temos uma situação em que o **cont** está sendo alterado de forma diferente a cada passo, ou seja, ele sempre dobra pois **step** é sempre o valor anterior de **cont**. Ainda temos o fato de o fator de parada estar sempre se modificando também. Então teremos:

i	cont	step	until	
1	1	1	3	
2	2	2	6	
3	4	4	9	
4	8	8	12	
5	16	16	15	→ fim do laço

ADA

```
CONT : FLOAT := 1.35;  
for CONT in 1..10 loop  
    SOMA := SOMA + CONT;  
end loop
```

Neste caso, a variável **CONT** do tipo **FLOAT** não é afetada pelo laço **for**. Após a finalização do laço, a variável **CONT** ainda é **FLOAT** com valor 1.35, pois será oculta pelo contador **CONT** do laço, que foi implicitamente declarado para ser o tipo da faixa discreta **INTEGER**

C, C++ e Java

```
for (ind = 0; ind <= 10; ind++)  
    soma += lista[ind];
```

Todas as expressões do **for** do C são opcionais, ou seja:

- Se a segunda expressão estiver ausente (ind = 0; ; ind ++), o laço será infinito.
- Se a primeira expressão estiver ausente, nenhuma inicialização está sendo feita.
- Se a terceira expressão estiver ausente, o contador não está sendo incrementado.

Outra observação que pode ser feita é que toda expressão pode conter múltiplas instruções:

```
for (cont = 0, cont2 = 0.0;  
    cont <= 10 && cont2 <= 100.0;  
    soma = ++cont1 + cont2, cont2 *= 2.5);
```

Observações:

- Como em C, valores diferentes de zero significam verdadeiro e valores iguais a zero são falso, temos que a segunda expressão pode ser uma expressão aritmética.
- No C++ é possível definir uma variável na primeira expressão e, seu escopo vai da definição até o final do laço. Ex.

```
for (int cont = 0; cont < comp; cont++) { ... }
```

- Em Java, a expressão de controle do laço se restringe ao tipo booleano.

VI.2.2 – Laços controlados logicamente:

São laço controlados por expressões booleanas, ao invés de contadores. Exemplo: **WHILE** e **REPEAT**. Podem apresentar pré-teste ou pós-teste. Programadores devem ter mais cuidado com o pós-teste, pois a sequência de instruções sempre será feita, pelo menos, uma vez.

PASCAL	C
WHILE	
<pre>soma := 0; readln (indat); while (indat >= 0) do begin soma := soma + indat; readln (indat); end;</pre>	<pre>soma = 0; cin >> indat; while (indat >= 0) { soma += indat; cin >> indat; }</pre>
REPEAT	DO – WHILE
<pre>readln (valor); repeat valor := valor /10; dígitos := dígitos +1; until (valor <= 1);</pre>	<pre>cin >> valor; do { valor /= 10; dígitos ++; } while (valor > 1);</pre>

VI.2.3 – Mecanismos de controle de laços localizados pelo usuário

Exemplo: EXIT, BREAK.

ADA => tem saídas condicionais e incondicionais

exit [rotulo_do_laço] [**when** condição]

C, C++ => saídas incondicionais não rotuladas (**break**)

JAVA => saídas incondicionais rotuladas (**break**)

Tanto o **EXIT** quanto o **BREAK** servem para saídas múltiplas de laços, principalmente em situações inesperadas. Mas, se mal utilizadas, diminuem a legibilidade do programa.

VI.3 – Desvio incondicional

Transfere o controle da execução para um lugar pré-determinado pelo programador.
 exemplo: goto

Esse tipo de desvio pode causar sérios problemas de legibilidade, tornando os programas pouco confiáveis e de difícil manutenção, apesar de ser bastante flexível.

VI.4 – Comandos protegidos:

```
IF  E1 -> C1
   E2 -> C2
   ...
   En -> Cn
FI
```

Onde E_i fornece um resultado lógico.

Qualquer E_i que seja verdadeiro provoca a execução de C_i. Esta forma foi sugerida por Dijkstra em 1975. Utilizados em programação concorrente (CSP e ADA).

Essa construção parece com seleção múltipla, mas sua semântica é diferente: Todas as expressões booleanas são avaliadas cada vez que se alcança esta construção durante a execução. Se mais de uma expressão for verdadeira, uma delas será escolhida não-deterministicamente. Se nenhuma for verdadeira, ocorrerá um erro que causará a finalização do programa.

Exemplos:

1) Encontrar o maior de dois números

```
if x >= y -> max := x
[ ] y >= x -> max := y
fi
```

2) Dados 4 números, organiza-los de forma que $q1 < q2 < q3 < q4$

```
do q1 > q2 -> temp := q1; q1 := q2 ; q2 := temp;
[ ] q2 > q3 -> temp := q2; q2 := q3 ; q3 := temp;
[ ] q3 > q4 -> temp := q3; q3 := q4 ; q4 := temp;
od
```

VI.5 Lista de Exercícios

1 – O que é uma estrutura de controle?

2 – Que problema pode ocorrer ao se fazer aninhamento de seletores bidirecionais? Dê uma sugestão de solução.

3 – O que é uma instrução de laço pré-teste? E pós-teste?

4 – Qual a característica da instrução **for** do ALGOL60 que torna os programas que a usam difíceis de serem lidos?

5 – Analise os potenciais problemas de legibilidade ao usar palavras reservadas de fechamento para instruções de controle que são o inverso das palavras reservadas iniciais.

6 – Reescreva o código abaixo usando uma estrutura de laço controlado por contador, em Pascal, ADA e Java, supondo que as variáveis são do tipo inteiro. Discuta qual a linguagem tem melhor capacidade de escrita, melhor legibilidade e melhor combinação das duas.

```
k := j;
laço:
  if k > 10 then goto fora
  k := k + 1
  i := 3 * k - 1
  goto laço
fora: ...
```

7 – Refaça o problema anterior transformando todas as variáveis e constantes em tipo real e mude a instrução

k := k + 1

para

k := k + 1.2

8 – Reescreva o código abaixo usando uma instrução de seleção múltipla em Java, supondo que todas as variáveis sejam do tipo inteiro.

```
if (k = 1) or (k = 2) then j := 2 * k - 1
if (k = 3) or (k = 5) then j := 3 * k + 1
if (k = 4) then j := 4 * k - 1
if (k = 6) or (k = 7) or (k = 8) then j := k - 2
```

9 – Considere a instrução abaixo em ALGOL 60:

```
for i := j + 1 step i * j while i <= 3*j do
    j := j + 1;
```

Suponha que o valor inicial de j seja 1. Liste a sequência de valores para a variável i usada, supondo que:

- a) Todas as expressões sejam avaliadas uma vez na entrada do laço
- b) Todas as expressões sejam avaliadas antes de cada iteração
- c) Expressões **step** sejam avaliadas uma vez na entrada do laço, expressões **while** são avaliadas antes de cada iteração.
- d) Expressões **while** sejam avaliadas uma vez na entrada do laço, expressões **step** são avaliadas antes de cada iteração, logo após incrementar o contador dos laços.

10 – Seja a instrução case do Pascal abaixo. Reescreva-a usando seleção bidirecional.

```
case ind of
    2, 4 : par := par + 1;
    1, 3 : impar := impar + 1;
    0 : zero := zero + 1;
    else erro := true;
end;
```

11 – Seja o código em Java abaixo. Reescreva-o sem usar **goto** nem **break**:

```
j = -3;
for (i = 0; i < 3; i++) {
    switch (j + 2) {
        case 3:
        case 2: j--; break;
        case 0 : j += 2; break;
        default : j = 0;
    }
    if ( j > 0 )
        break;
    j = 3 - i;
}
System.out.println (i);
```

12 – Seja o código abaixo que localiza a primeira linha da matriz $n \times n$, de elementos inteiros, chamada *mat*, que só tem elemento zero. Reescreva o código sem *goto* nem *break*, em PASCAL ou Java:

```
for i := 1 to n do
begin
  for j := 1 to n do
    if mat[i,j] <> 0 then goto rejeita
  writeln ('Primeira linha so com zeros : ', i);
  break;
rejeita:
end;
```

13 – Reescreva o código abaixo, em Pascal ou C++, utilizando um comando de repetição pós-teste, sem operadores unários nem ternários:

```
cin >> x; //comando de entrada de dado pelo teclado
cin >> y;
while (x % 3 != 0 ) {
  (x > y) ? x = y++ : x *= ++y;
}
```

CAPÍTULO VII - ABSTRAÇÃO DE PROCESSOS

VII.1 – Introdução

Abstraindo computacionalmente etapas para que uma determinada tarefa seja executada.

Ex. de um subprograma: procedimento, função.

Características gerais dos subprogramas utilizados:

- Possuem um único ponto de entrada
- Um único subprograma em execução a cada instante
- O controle sempre retorna ao chamador quando a execução do subprograma encerra-se

VII.2 – Definições

- **Subprograma:** descreve a interface e as ações
- **Chamada:** é a solicitação explícita para executar o subprograma
- **Subprograma ativo:** se for chamado e sua execução não foi encerrada ainda.
- **Cabeçalho:** é a primeira linha de definição do subprograma. Especifica o subprograma (função ou procedimento), fornece um nome ao subprograma, e pode especificar a lista de parâmetros.
- **Perfil do parâmetro:** é o número, a ordem e os tipos de seus parâmetros formais
- **Protocolo:** é seu perfil de parâmetros mais seu tipo de retorno (se for função)
- **Argumento:** são os valores passados para a abstração, é o valor armazenado no parâmetro real.

Exemplo:

```
PROCEDURE Teste (var a: integer);  
BEGIN  
    WRITELN (´DIGITE O VALOR DE A:´);  
    READLN (A);  
END;
```

```
void Teste (int *a)  
{  
    printf(%, "DIGITE O VALOR DE A:");  
    scanf (A);  
}
```

Os subprogramas podem ter declarações e definições:

Declarações => protótipos => cabeçalho => informações sobre tipos, sem definir variáveis. São necessárias para verificação estática de tipos.

Definições => o subprograma todo

VII.3 – Parâmetros

Há duas maneiras de um subprograma acessar dados: **variáveis globais** ou **os parâmetros**. Os dados passados por parâmetro são acessados por nomes locais. A passagem de parâmetros é mais flexível do que utilizar variáveis globais. Assim, pode-se

parametrizar um processo, onde será aceita qualquer variável do tipo do parâmetro. O acesso extensivo a variáveis globais pode reduzir a confiabilidade.

VII.3.1 – Tipos de Parâmetros

➤ Parâmetros formais

```
procedure teste (a : real);
```

São os parâmetros do cabeçalho do subprograma. Muitas vezes recebem o nome de variáveis fictícias.

➤ Parâmetros reais

```
teste ( b );
```

São os parâmetros passados quando o subprograma é chamado.

Existem duas formas básicas de vinculação do parâmetro formal com o real

➤ Parâmetro posicional (PASCAL, C, C++, Java, etc.)

Quando a vinculação entre os parâmetros é realizada baseando-se em sua posição no cabeçalho.

Desvantagem: lembrar ordem em uma lista grande de parâmetros.

➤ Parâmetro nomeado (ADA, FORTRAN)

O nome do parâmetro formal ligado ao real é especificado.

```
procedure somador ( m_vetor : tp_vet; m_comp: integer);  
    somador (c=>m_comp, v =>m_vetor);
```

Desvantagem: lembrar os nomes dos parâmetros formais

Existem ainda LP's, como C++, ADA, FORTRAN 90, em que parâmetros formais podem ter valores **padrão**. Isso quer dizer que, se nenhum parâmetro real for passado, o subprograma utilizará o valor padrão.

```
float calc_pág (float renda, float tarifa_imposto, int isencoes = 1)  
    pagamento = calc_pág (20000.0, 0.15);
```

Em C++, os parâmetros são organizados de modo que o padrão venha por último, por que são posicionais.

VII.3.2 – Métodos de passagem de parâmetros

Passagem de parâmetros é a maneira de transmitir parâmetros aos subprogramas chamados. São caracterizados por:

- 1) receber dados do parâmetro real - *modo de entrada*
- 2) transmitir dados ao parâmetro real - *modo de saída*
- 3) ambos - *modo de entrada e saída*

a) Passagem por valor

O parâmetro real é utilizado para inicializar o formal, que é uma variável local ao subprograma.

Devantagens: custoso (armazenamento adicional e tempo para transferência do parâmetro real para o formal)

b) Passagem por resultado

O parâmetro real não transmite nada como entrada, mas recebe algo no término do subprograma.

O parâmetro local age como uma variável local, mas seu valor é transferido para o parâmetro real ao terminar o subprograma. O valor inicial do parâmetro real não deve ser utilizado pelo subprograma.

Devantagens: Custo (armazenamento e operações cópias extras), colisão com parâmetros reais.

sub (p1, p1);

Neste caso, os parâmetros formais têm nomes diferentes, mas o mesmo tipo. Então, os parâmetros reais podem ser iguais. Mas, se um deles for passado por resultado, pode haver problemas.

c) Passagem por valor-resultado

É uma mistura de passagem por valor e passagem por resultado. Os valores reais são transferidos para os formais. No final do subprograma, o valor formal é transferido de volta para o real. O parâmetro formal age como variável local.

Também é chamada de passagem por cópia.

Devantagens: Custo (armazenamento múltiplo para parâmetros e tempo para cópias de valores), colisão (devido à ordem com que os parâmetros são atribuídos).

d) Passagem por referência

Transmite um caminho de acesso, um endereço, por exemplo, que permite o acesso ao próprio parâmetro real.

Vantagem: menos tempo, economia de memória.

Devantagem: acesso aos parâmetros formais – lento, pode ocasionar alteração indesejada, podem ocorrer apelidos, por exemplo:

void fun (int *primeiro, int *segundo)

com a seguinte chamada:

fun (&total, &total)

primeiro e segundo em fun serão apelidos.

fun (&lista[i], &lista[j])

mas, e se $i = j$?

e) Passagem constante

O parâmetro real é associado ao parâmetro formal diretamente (através de seu endereço, por exemplo), mas não pode ser alterado.

```
void fun (const int &p1, int p2, int &p3) {...}
```

Observações:

- Passagem constante e parâmetros em modo de entrada não são a mesma coisa, pois parâmetros em modo de entrada podem ser alterados no subprograma, mas essas alterações não são refletidas nos parâmetros reais correspondentes. Parâmetros constantes nunca podem ser alterados.
- Quando o acesso à definição dos parâmetros é mais lento que o acesso por cópia? Quando a execução do programa se dá de forma distribuída em máquina remota, a chamada está em uma máquina, mas a execução do subprograma se dá em outra máquina.

VII.3.3 Verificação de tipos dos parâmetros

É a verificação da coerência do parâmetro real com o formal. Sem essa verificação, o programa pode gerar erros difíceis de diagnosticar.

O FORTRAN 77 - não exige verificação de tipos

Pascal, Modula-2, FORTRAN 90, Java e Ada exigem.

C original não fazia:

```
void funcao( x );  
int x;  
{ }
```

No ANSI C pode ou não fazer:

```
void funcao ( int x );  
{ }
```

ou do outro modo.

VII.4 – Módulo

Qualquer unidade de programa que é nomeada e que pode ser implementada como entidade independente.

Um módulo pode ser um simples procedimento ou função, ou um grupo de vários componentes (tipos, constantes, variáveis, procedimentos, ...) declarados com um objetivo comum. Um módulo **encapsula** seus componentes.

VII.5 Abstração de Dados – Tipos Abstratos de Dados

É o mecanismo através do qual observa-se o domínio de um problema e foca-se nos objetos, ações e propriedades, que são relevantes para uma aplicação específica.

Descreve as características essenciais de um objeto que o distingue de todos os outros tipos de objetos.

Proporciona uma abstração sobre uma estrutura de dados.

- Código e estrutura de dados de uma abstração devem estar armazenados em um mesmo local
- Proporciona o ocultamento da informação, promovendo proteção contra acessos inesperados à estrutura de dados.
- tipo + ocultamento da informação + encapsulamento = TAD

Os tipos abstratos de dados são constituídos de duas partes: Especificação (interface) e a Implementação

Vantagens:

Legibilidade, facilmente modificado, mais seguro

Exemplos:

```
program objeto;
type
    reg = OBJECT
        X,Y: INTEGER;
        PROCEDURE INICIAR ( a,b:integer);
    END;

PROCEDURE REG.INICIAR ( a,b:integer);
begin
    X := A;
    Y := B;
END;

var
    R : REG;

BEGIN
    R.INICIAR (10,20);
END.
```

Outro exemplo:

```
public class No
{
    String info;
    No prox;
    No (Item _info){
        this.info = _info;
    }
}

public class Lista{
    private No prim;
    private int NElem;
    public Tabela(){
        this.prim = null;
    }
}
```



```
public boolean éVazia () {...}
public int getNElem() {...}
public void inserePrimeiro(String elem){...}
public boolean remove (String elem){...}
}
public class UsaLista{
    public static void main (String[] args){
        lista p;

        p.inserePrimeiro("elemento 1");
        p.inserePrimeiro("elemento 2");
        p.remove("elemento 1");
    }
}
```

VII.6 Subprogramas Sobrecarregados (*OVERLOADING*)

Um identificador ou operador é dito estar sobrecarregado (overloaded) se este simultaneamente denota dois ou mais subprogramas diferentes.

Em **PASCAL** apenas operadores e subprogramas pré-definidos são sobrecarregados, não é permitido ao programador criar subprogramas sobrecarregados.

Exemplo: `writeln ('digite algo');`
`writeln (a); {a inteiro}`

C++, ADA, JAVA.

Obs. : Cada versão do subprograma deve ser diferente quanto ao número, à ordem ou aos tipos de seus parâmetros.

Muitas vezes o overloading é chamado de **polimorfismo *ad hoc***

Exemplo em C++:

```
#include <stream.h>
int maior (int i, int j)
{
    return ( i > j ? i : j);
}

double maior (double i, double j)
{
    return (i > j ? i : j );
}

main ( )
{
    int a = 10,    b = 15, c;
    double d1 = 7.0, d2 = 14.5, d3;

    c = Maior (a,b);
    d3=Maior (d1,d2);
    cout << form("Comparando %d    e %d, o maior e %d\n", a, b, c);
    cout << form("Comparando %f    e %f, o maior e %f\n", d1, d2, d3);
}
```

VII.7 – Subprogramas Polimórficos – Polimorfismo paramétrico

Um subprograma genérico ou polimórfico leva parâmetros de diferentes tipos em várias ativações, ou seja, um subprograma não teria os tipos de seus parâmetros definidos em tempo de compilação.

Exemplo: Um procedimento `inserePrimeiro` que seria genérico, serviria para inserir elementos em um lista de inteiros, em uma lista de real, etc...

ADA, C++, JAVA

Subprogramas polimórficos são chamados de polimorfismo paramétrico.

Exemplo em C++:

```
#include <stream.h>
template <class Tipo>
Tipo Maior (Tipo primeiro, Tipo segundo)
{
    return ( primeiro > segundo ? primeiro : segundo);
}

main ( )
{ int a = 10, b=15, c;
  double d1 = 7.0, d2 = 14.5, d3;

  c = Maior (a,b);
  d3= Maior (d1, d2);
  cout << form("Comparando %d e %d, o maior e %d\n", a, b, c);
  cout << form("Comparando %f e %f, o maior e %f\n", d1, d2, d3);
}
```

VII.8 Classes

É um novo modelo de dados definido pelo usuário, proporcionando encapsulamento, proteção e reutilização utilizando herança. Uma classe tipicamente contém:

- Uma especificação de herança
- Uma definição para a representação de dados
- Definição para as operações

Herança: É um mecanismo para derivar novas classes a partir de classes existentes. Uma classe derivada herda a representação de dados e operações de sua classe base, mas pode adicionar novas operações, estender a representação de dados ou redefinir a implementação de operações já existentes.

Assim poderíamos dizer que:

Classe = TAD + herança

Exemplo em Java:

Classe que define o objeto Pessoa que possui como atributos os dados que definem uma pessoa:

```
public class Pessoa {
    private String nome;
    private int identidade;
    private Data nascimento;
```

```
Pessoa (String n, int i, Data nasc){
    this.nome = n;
    this.identidade = i;
    this.nascimento = nasc;
}

public String toString(){
    return "Nome : "+this.nome+"\n Identidade : "+
           this.identidade+"\nData de nascimento : "+
           this.nascimento.toString();
}

public void setNome (String novoNome){
    this.nome = novoNome;
}

public String getNome (){
    return this.nome;
}
}
```

Classe que define o objeto Funcionário que é um tipo de Pessoa. Assim, um Funcionário possui os atributos de uma Pessoa (que são herdados, embora seus campos não possam ser diretamente acessados pois são privados) e outros atributos a mais. Assim como os atributos, a classe Funcionário também herda os métodos da classe Pessoa.

```
public class Funcionario extends Pessoa{
    private Data admissão;
    private float salario;

    Funcionario (String nome, int id, Data nasc, Data adm,
                 float sal){
        super (nome, id, nasc);
        this.admissão = adm;
        this.salario = sal;
    }

    public String toString(){
        return (super.toString()+"\n" + "Data de admissão : " +
               this.admissão.toString()+"\n" + "Salário : "+
               this.salario);
    }
}
```

Classe UsaPessoa, onde são declaradas duas variáveis: Uma do tipo Pessoa e outra do Tipo Funcionário (herdeiro de Pessoa)

```
public class UsaPessoa {
    public static void main (String[] args){
        Pessoa p1 = new Pessoa("Joao", 12345,
                               new Data ((byte)12,(byte)8,(short)1998));
        Funcionario f1 = new Funcionario ("Jose", 13243,
                                           new Data ((byte)30,(byte)8, (short)1980),
                                           new Data ((byte)02, (byte)3,(short)2003), 880.00f);
        System.out.println ("O nome do Funcionário é "+
                             f1.getNome());
        System.out.println ("O nome da Pessoa é "+p1.getNome());
        System.out.println ("Funcionario : "+f1.toString());
        System.out.println ("Pessoa : "+p1.toString());
    }
}
```

No console aparecerá as seguintes mensagens:

```
O nome do Funcionário é Jose
O nome da Pessoa é Joao
Funcionario : Nome : Jose
Identidade : 13243
Data de nascimento : 30/08/1980
Data de admissão : 02/03/2003
Salário : 880.0
Pessoa : Nome : Joao
Identidade : 12345
Data de nascimento : 12/08/1998
```

VII.9 Lista de Exercícios

- 1 – Quais as três características gerais de um subprograma?
- 2 – O que significa um subprograma estar ativo?
- 3 – O que é um perfil de parâmetro? O que é um protocolo de subprograma?
- 4 – O que são parâmetros formais? E parâmetros reais?
- 5 – Quais são as vantagens e desvantagens dos parâmetros de palavra-chave?
- 6 – Quais são os três modelos semânticos de passagem de parâmetros?
- 7 – Quais são os métodos de passagem de parâmetros, suas vantagens e desvantagens?
- 8 – Como podem ocorrer apelidos com passagem de parâmetros por referência?
- 9 – O que é um subprograma sobrecarregado?
- 10 – O que é polimorfismo paramétrico?
- 11 – Considere o seguinte programa escrito na sintaxe C:

```
void principal () {
    int valor = 2;
    int[] lista = {4, 3, 5, 7, 9};
    troca (valor, lista[0]);
    troca (lista[0], lista[1]);
    troca (valor, lista[valor]);
}
void troca (int a, int b) {
    int temp;
    temp = a;
    a = b;
    b = temp;
}
```

Para cada um dos métodos de passagem de parâmetros (passagem por valor, por referência, por valor-resultado), quais são todos os valores das variáveis **valor** e **lista** depois de cada uma das três chamadas de **troca**?