LINGUAGENS DE PROGRAMAÇÃO

Ismael H F Santos

1 Bibliografia

- Nunes, M. G. V.; Faceli, K., Tópicos em Linguagens de Programação, ICMC / São Carlos - USP, 1998

- Pratt, T.W., Programming Languages: Design and Implementation, 3ª edição, Prentice-Hall, 1996.

- Neto, Rangel M. L. J., Projeto de Linguagens de Programação, IX CSBC, VIII JAI, SBC, 1989.

- Ghezzi, C., Jazayeri, M. Programming Language Concepts 3ª Edition, New York: John Willey *&* Sons, 2000.

2 Considerações Gerais

2.1 Objetivo

- Introduzir e discutir os conceitos relacionados a Linguagens de Programação, principalmente no contexto de linguagens imperativas, e apresentar as principais características dos demais paradigmas de Linguagens de Programação: ***Imperativo ou Procedural; Funcional; Lógico e Orientado a Objeto.***

2.2 A quem se destina

- Alunos que desejem aprofundar seus conhecimentos sobre Linguagens de Programação.

3 Introdução

3.1 Definição de Linguagem de Programação

Uma Linguagem de Programação (LP) é uma notação especializada, a ser usada por uma ou mais pessoas, para expressar um processo (programa) através do qual um computador pode resolver um problema.

Uma LP é assim uma das ferramentas disponíveis para um programador implementar programas de computador. Com uma LP, o programador cria o código fonte de um programa, que irá sofrer um processo de ***tradução*** para gerar o código executável correspondente.

A LP escolhida para a implementação de um programa (ou conjunto de programas) deve suportar os critérios e requisitos do projeto desse programa, oferecendo facilidades que fazem dessa LP uma LP conveniente para a implementação do projeto, de forma fácil, segura e eficiente.

A seguir apresentamos as principais razões para o estudo das LPs:

- ***Adquirir conhecimentos apropriados para a implementação de algoritmos elegantes e eficientes;***
- ***Compreender o papel da LP no processo de desenvolvimento de Software (SW) e comparar os diferentes mecanismos disponíveis nas LPs para a construção de programas;***
- ***Conhecer as diversas categorias de LPs e como elas se relacionam para saber escolher a LP apropriada para cada aplicação;***
- ***Facilitar o aprendizado de novas LPs aprendendo os conceitos básicos e mais importantes que em geral estão presentes em muitas LPs.***
- ***Facilitar o Projeto de novas LPs. Apesar da maioria dos programadores não ter que projetar novas LPs, ainda assim todo programa tem uma interface com o usuário, que de fato é uma forma de Linguagem de Programação através da qual o usuário se comunica com a aplicação. Assim muitos projetos de software complexos como um Editor de Textos, um Sistema Operacional, um Pacote Gráfico, demandam um projeto de interface que apresentam os mesmos problemas encontrados durante o Projeto de uma nova LP.***

3.2 Processo de Desenvolvimento de Software e o Projeto de LPs

Alterações significativas se processaram na história da computação, e com o barateamento do hardware (HW), devido em grande parte ao surgimento do microcomputador, as atenções da indústria tem se voltado para o software. Máquinas mais complexas e poderosas exigem uma programação mais sofisticada, para que seu potencial possa ser explorado. A necessidade de aplicações mais amigáveis e intuitivas têm exigido projetos de software cada vez mais complexos. Devido a esta crescente complexidade e o fato da produção de software estar ainda em um estágio primitivo, sendo ainda uma atividade essencialmente manual, muito esforço tem sido feito para tentar se diminuir os custos da produção de software.

Dessa forma, diversas **Metodologias de Desenvolvimento** e novas **LPs** tem sido propostas para facilitar o processo de desenvolvimento de software. O Modelo mais usado para compreender o processo de desenvolvimento de software é o **Ciclo de Vida**, que descreve de forma simplificada as fases da história de vida de um programa, ou sistema (conjunto de programas) :

1. **Analise a Especificação de Requisitos** - Nesta fase se procura definir com clareza qual é o problema considerado, e que características deve ter uma solução computacional (HW e SW), para que essa solução atenda as necessidades do cliente. São elaborados os seguintes documentos: ***Estudo de Viabilidade e Custo, Especificação de Requisitos.***

2. **Projeto e Especificação do Sistema** - A partir dos documentos gerados na fase anterior os projetistas de software elaboram a ***Especificação do Projeto do Sistema***, identificando todos os módulos que compõem o sistema e suas interfaces com outros sistemas e dispositivos. A metodologia de projeto utilizada nesta fase pode ter um grande impacto da escolha da LP a ser usada na implementação do sistema.

3. **Implementação ou Codificação** - Nesta fase o SW é efetivamente escrito em uma ou mais LPs escolhidas para isso. Após a confecção de cada módulo ele é testado para garantir que satisfaça as suas especificações de interface.

4. **Certificação ou Validação** - Nesta fase o sistema é testado para verificar se os requisitos especificados estão sendo atendidos.

5. **Manutenção** - Após a entrega do sistema, alterações podem se tornar necessárias quer seja para correção de erros quer seja para se adicionar novas funcionalidades para torna-lo mais adequado para a aplicação, cujas necessidades já podem ser diferentes daquelas consideradas na fase de especificação de requisitos.

Naturalmente, cada uma das fases mencionadas pode ser várias vezes repetida, até que o produto final seja considerado satisfatório. Atualmente a tendência tem sido o desenvolvimento incremental ou por **Prototipagem**, onde são feitas sucessivas iterações por todas as fases mencionadas acima. Dessa forma o usuário toma contato com o sistema o mais cedo possível e novos rumos podem ser tomados para se garantir o sucesso do sistema.

As primeiras LPs foram projetadas para *programação* e não para o *desenvolvimento de sistemas*. Com o objetivo de aumentar a produtividade na produção de software foi criado a idéia do "**ambiente de desenvolvimento**" que se constitui em um conjunto integrado de ferramentas e técnicas para auxiliar no desenvolvimento de sistemas em todas as fases do **Ciclo de Vida**. As ferramentas utilizadas para a fase de implementação são: Linguagens de Programação, Editores de Texto, Compiladores, Ligadores, etc.

3.2.1 Relação entre Metodologia de Desenvolvimento e Projeto de LPs

No que concerne a **Metodologia de Desenvolvimento** a **LP** deve, de preferência, prover mecanismos para dar suporte a mesma filosofia. **Metodologias de Desenvolvimento Top-Down** requerem **LPs** que permitam representar níveis hierárquicos utilizados no projeto com a LP, enquanto **Metodologias de Desenvolvimento Bottom-Up** requerem **LPs** que permitam representar partes básicas do sistema de forma modularizada. Linguagens mais antigas, como **FORTRAN** (Formula Translation) não foram projetadas para suportar metodologias específicas de projeto. Por outro lado, **Pascal** foi projetado com o objetivo de suportar o desenvolvimento **Top-Down** e programação estruturada.

Metodologias de Desenvolvimento de sistemas e **Projeto de Linguagens de Programação** devem se completar, e com o passar do tempo estão se tonando realidade. O exemplo mais claro é o caso da **Orientação a Objeto** (AOO e POO - Análise e Projeto Orientado a Objetos) onde se utiliza a **Ocultação de Informação (Infomation-Hiding)** como **Metodologia de Desenvolvimento** e **Abstração de Dados**, como princípio de **Projeto de Linguagem de Programação.**

Na fase de projeto (AOO/POO), o sistema é decomposto em módulos (objetos), utilizando-se a **Ocultação de Informação**, onde somente o próprio módulo (Objeto) conhece e pode consultar as suas informações. O módulo (objeto) fornece funções de acesso que podem ser usadas pelos outros módulos (objetos) para consultar ou atualizar a informação contida na sua estrutura de dados interna. Na fase de codificação, Linguagens que suportam a **Abstração de Dados** como SIMULA 67, ADA, C++, Smalltalk, Java, etc são capazes de implementar facilmente esta metodologia.

3.2.2 Características desejáveis de uma Linguagem de Programação

Os objetivos de **Projeto de Linguagens de Programação** impostos pelo processo de desenvolvimento de sistemas podem ser sumariados em três objetivos principais, a saber:

- **Confiabilidade** - Os programas especificados devem ser confiáveis: A linguagem não deve induzir o programador a erros, em particular erros que não possam ser descobertos com facilidade.

- **Manutenibilidade** - Os programas devem ser manuteníveis: A linguagem deve ajudar o programador a corrigir erros, facilitando a sua identificação e a determinação da ação de correção, seja qual for a causa original do erro.

- **Eficiência** - Os programas devem rodar eficientemente: A estrutura da linguagem deve facilitar o processo de geração e otimização de código, permitindo a construção de compiladores que gerem código objeto que faça uso eficiente dos recursos de máquina disponíveis.

Os objetivos de **Confiabilidade e Manutenibilidade** dos programas são favorecidos pelas seguintes qualidades de uma LP:

1. **Legibilidade** - Facilidade que a linguagem de programação oferece para que um programador leia e compreenda um programa sem encontrar ambigüidades. Tal característica facilita testes e correções pelo programador. Um programa em processo de alteração pode ter sido escrito há algum tempo, ou por outra pessoa que não se encontra disponível para consulta, e por isso uma boa documentação e uma utilização adequada de comentários são importantes, por mais legível que seja o texto de um programa.

2. **Redigibilidade** - Facilidade de escrita de um programa de uma maneira que seja natural para o problema. Detalhes e truques de linguagem não devem desviar a atenção do programador da atividade mais importante que é resolver o problema. Apesar de ser um critério subjetivo, podemos dizer que ***Linguagens de alto nível*** são mais ***Redigíveis*** que ***Linguagens de baixo nível***. Esta propriedade é de certa forma conflitante com a legibilidade pois a medida que menos se escreve aumenta a **Redigibilidade** mais diminui a **Legibilidade.**

O exemplo a seguir mostra o caso do comando IF do Pascal. Apesar de Pascal ser

considerada uma linguagem bastante legível é necessário examinar com cuidado os trechos de código da figura-1 abaixo para saber se o código da versão 1 é equivalente ao código da versão 2, ao código da versão 3 ou ao da versão 4.

Pascal-1	Pascal-2	Pascal-3	Pascal-4

`if x>1 then`
`  if x=2 then`
`  x:=3`
`else`
`  x:=4;`
	`if x>1 then`
	`  if x=2 then`
	`  x:=3`
	`  else`
	`  x:=4;`
		`if x>1 then`
		`  if x=2 then`
		`  x:=3;`
		`  else`
		`  x:=4;`
			`if x>1 then`
			`BEGIN`
			`  if x=2 then`
			`  x:=3;`
			`END`
			`else`
			`  x:=4;`

Figura-1 Legibilidade no PASCAL

Na figura-2 damos exemplos, em diversas Linguagens, onde a **Legibilidade** pode ser melhorada pelo acréscimo de novas marcas usadas para indicar o final da construção sintática IF, ao custo, portanto, de uma diminuição da **Redigibilidade.**

C/C++	Algol	ADA	Fortran

`if ( x>1 )`
`{`
`  if( x==2 )`
`  x=3;`
`}`
`else`
`  x=4;`
	`if x>1 then`
	`  if x=2 then`
	`  x=3;`
	`  else`
	`  x=4;`
	`  fi`
	`fi`
		`if x>1 then`
		`  if x=2 then`
		`  x:=3;`
		`  else`
		`  x:=4;`
		`  end fi;`
		`end fi`
			`if (X.GT.1) then`
			`  if (X.EQ.2) then`
			`  x=3`
			`  else`
			`  x=4`
			`  endif`
			`endif`

Figura-2 Legibilidade x Redigibilidade em outras Linguagens

Uma outra qualidade importante de uma LP é a Manutenibilidade que definimos como:

3. **Manutenibilidade** - facilidade de modificação dos programas para incorporar novos requisitos de projeto e com baixo custo. Apesar de ser também uma qualidade subjetiva é possível, porém, identificar algumas propriedades que tornam um programa mais **manutenível**. Por exemplo, diversas linguagens permitem que constantes recebam nomes simbólicos. Uma escolha apropriada de nomes para as constantes aumenta a **Legibilidade** além de permitir uma parametrização do código que permitirá que ele seja rapidamente alterado em caso de necessidade. Vejamos então o próximo exemplo :

C/C++	PASCAL	Fortran

```
#define NPALS 65536

int mem[NPALS];


void AlocaMem( void )
{
 int adress;

 if( adress > NPALS )
  printf(
    "\nmemória out!");
}
```

```
CONST
 NPALS = 65536;
VAR   Mem: array
  [1..NPALS] of integer;

Procedure ALOCA_MEM;
VAR
 Adress: 1.. NPALS;
BEGIN
 If adress > NPALS then
  Writeln(
     'memoria out!');
END;
```

```
PARAMETER (NPALS=65536)
INTEGER mem(NPALS)
COMMON /MEMORY/ mem


SUBROUTINE ALOCA_MEM
INTEGER mem(NPALS)
COMMON /MEMORY/ mem
INTEGER adress

IF(adress.GT.NPALS)THEN
  Print*,'memoria out!'
ENDIF
RETURN
END
```

Figura-3 Mecanismos de linguagem que aumentam a Manutenibilidade

Outras qualidades importantes que caracterizam uma boa **LP** são:

4. **Simplicidade** - a LP deve prover um número mínimo de conceitos e estruturas, de fácil aprendizado, com pouco risco de má interpretação. Os conceitos devem ser naturais, com representação única e "clara" de forma que possam ser usados como primitivas durante o desenvolvimento de novos algoritmos.

5. **Suporte para Abstração** - apenas aspectos relevantes aos **TAD (Tipo Abstrato de Dados)**. Tem reflexo nas tarefas de design, implementação e modificação de programas. No nível de dados o programador pode trabalhar mais efetivamente usando abstrações mais simples, que não incluam detalhes irrelevantes dos dados. No nível de procedimentos a abstração facilita modularidade e boas práticas de design.

6. **Expressividade** - facilidade com que um **TAD** pode ser representado. As LPs devem permitir uma representação natural de objetos e procedimentos (por exemplo, estruturas de dados e de controle apropriadas). Simplicidade e Expressividade são características um tanto antagônicas e a maior aplicação de uma ou outra depende do domínio de aplicação do problema. Deste modo deve-se restringir os problemas a domínios específicos.

7. **Ortogonalidade** - refere-se à integração entre os conceitos, o grau de interação entre diferentes conceitos, e como eles podem ser combinados de maneira consistente. Por exemplo, quando uma "string" não pode ser passada como parâmetro, os conceitos de "string" e "parâmetro" não podem interagir, e portanto, há falta de ortogonalidade. Além disso, se uma LP usa o operador := para atribuição de "inteiro" e <- para atribuição de "string", então o conceito "atribuição interage inconsistentemente com os de "inteiro" e "string". A ortogonalidade reduz o número de exceções de regras e torna a LP mais fácil de ser aprendida. Também leva a dificuldades: combinações difíceis de se implementar ou mesmo improváveis de ocorrer.

8. **Portabilidade** - movimento de um programa de uma máquina a outra. Utilização de um padrão independente de máquina.

3.3 Paradigmas das Linguagens de Programação

As LPs surgiram da necessidade de livrar o programador dos detalhes mais íntimos das máquinas em que a programação é feita, permitindo a programação em termos mais próximos do problema em um nível de abstração mais alto. **Linguagens de Montagem** (Assembly Languages) oferecem essencialmente a utilização de nomes em vez de endereços e de códigos (mnemônicos) de instruções, mas, sendo especificas, não oferecem nenhuma facilidade para transporte de programas para outras maquinas.

Linguagens como **Fortran** oferecem uma certa independência de máquina, porém o **Fortran** ainda segue de perto a estrutura das Linguagens de Montagem, e algumas de suas instruções refletem a estrutura das maquinas em que a linguagem foi inicialmente implementada. A LP **Algol-60** embora derivada do **Fortran**, foi projetada de forma independente da estrutura das máquinas existentes, e sua implementação leva ao desenvolvimento de novas técnicas de compilação, e mesmo, mais tarde, de novas Arquiteturas de Computadores.

A programação estruturada apareceu inicialmente com o objetivo de organizar o projeto de programas, através do uso de estruturas de comandos padronizadas e mais legíveis.Com a evolução das técnicas iniciais de estruturação de programas, surgiram técnicas como a construção de **Tipos Abstratos de Dados (TAD)** e a programação orientada a objetos (**POO**), que podem ser consideradas como "programação estruturada moderna"

As Linguagens Lógicas (Prolog) e as Funcionais (LISP e ML) tiveram gênese independentes e estão baseadas em teorias matemáticas distintas. As Linguagens Lógicas, como Prolog, se baseiam no Calculo de Predicados da Lógica Matemática e as Linguagens Funcionais, como o LISP, na teoria Lambda Calculus

Atualmente temos as seguintes classificações para os diferentes paradigmas em que as diversas linguagens de programação se baseiam:

1. Linguagens Imperativas ou Procedurais- caracterizam-se por um processamento basicamente seqüencial e uso intenso da atribuição. As variáveis descrevem o estado da computação a cada instante. São linguagens apropriadas para máquinas cuja arquitetura é baseada na máquina de Von Newmann. Podemos dividir as Linguagens imperativas em dois tipos:

1.1. Estrutura Procedural Simples - Fortran, Cobol, Algol 60, APL, Basic, PL/I, Algol 68, C;

1.2. Estrutura Procedural em Blocos - Pascal, Módula 2 e

2. Linguagens Baseadas em Regras ou Lógicas - a computação é realizada por um motor de inferência que a partir de um conjunto de regras fornecido (programa), e uma pergunta feita pelo usuário tenta-se verificar a veracidade ou não da pergunta. ***Prolog (PROgraming in LOGic)*** é a linguagem mais famosa deste paradigma. Outros exemplos são ***YACC(Yet Another Compiler Compiler).***

3. Linguagens Aplicativas ou Funcionais - neste paradigma em vez de nos concentrarmos na *seqüência de transições de estados que a computação* deve realizar até obtermos uma resposta, nós nos concentramos na seqüência de funções que devem ser aplicadas aos dados para que a partir do estado inicial nos obtenhamos uma resposta. Exemplos de linguagens deste paradigma são o ***Lisp*** e o ***ML***;

4. Linguagens Orientadas a Objeto - neste paradigma objetos complexos são construídos, e um conjunto limitado de operações sobre estes objetos são definidas. Novos objetos mais complexos podem ser construídos a partir dos anteriores. De uma certa forma neste paradigma nós estamos tentando fundir o melhor do paradigma Imperativo com o melhor do paradigma Funcional. Exemplos de linguagens que implementam este paradigma são o ***Simula 67, Ada, Smalltalk, C++, Java, Eiffel etc;***

3.4 Perspectiva Histórica das Linguagens de Programação

A seguir falamos um pouco da história de algumas Linguagens de Programação que introduziram conceitos importantes para as futuras LPs e que ainda estão em uso. Essas linguagens estão classificadas em períodos, de acordo com a época em que surgiram.

1956 - 1965

- FORTRAN (FORmula TRANslation)
- COBOL (COmmon Business Oriented Language)
- ALGOL 60 (ALGorithmic Oriented Language)
- LISP (LISt Processing)
- APL (A Programming Language)
- BASIC (Beginners All-purpose Symbolic Instruction Code)

1966 - 1970 (LPs baseadas em ALGOL)

- PL/I (Programming Language I)
- SIMULA 67
- ALGOL 68
- PASCAL

1971-1980 (criadas na década de 70)

- Scheme
- PROLOG (PROgramming in LOGic)
- SMALLTALK
- C
- MODULA 2
- ADA

1981-1990 (criadas na década de 80)

- SMALLTALK-80
- PostScript
- FORTRAN 90
- C++
- SML (Standard Markup Language)

1991-1995

- Linguagens Visuais: VB, Delphi
- TCL
- PERL
- EIFFEL
- ML
- JAVA

3.4.1 FORTRAN (FORmula TRANslation)

A linguagem Fortran, desenvolvida em 1956 por John Backus, foi proposta visando a resolução de problemas científicos, para isto utilizando a notação algébrica. Foi desenvolvida, inicialmente para uma máquina específica, o IBM 704. A primeira padronização da linguagem foi adotada em 1966, e uma grande revisão foi feita nos anos 70 originando o padrão FORTRAN 77 que é o mais usado, nos dias de hoje, no meio científico.

Para acompanhar a evolução do HW e das Linguagens de Programação uma nova revisão do padrão foi feita no início dos anos 90, originando o padrão FORTRAN 90, que representa uma mudança radical na estrutura da linguagem criando o conceito de Módulos (para implementação de TAD), estruturas (record), regras de escopo, definição de novos tipos pelo usuário, recursividade, arrays dinâmicos, ponteiros e alocação dinâmica de memória

Um dos motivos pelos quais o FORTRAN é ainda muito utilizado é a disponibilidade de uma vasta biblioteca de software contendo rotinas freqüentemente utilizadas, tais como rotinas para cálculo de funções trigonométricas, equações algébricas polinomiais, etc, o que permite uma redução dos custos e tempo de desenvolvimento dos programas.

Contribuições para futuras linguagens:

- ***variáveis***
- ***comando de atribuição***
- ***conceito de tipos***
- ***modularidade (com o uso de Subprogramas)***
- ***E/S formatadas***
-

Exemplos de programas FORTRAN:

1. **Programa HelloWorld**

```
*       - Programa Principal -
        Program HELLO_WORLD
        CALL PRINTMSG
        STOP
        END
        SUBROUTINE PRINTMSG
        PRINT*,'Hello World'
        RETURN
        END
```

2. **Programa Converte Decimal para Binário**

```
*       Programa que converte um valor decimal, maior ou igual a zero,
*       lido através do console do sistema, em seu correspondente valor
*       binário, imprimindo este valor na tela do console.
*       Autores: José Carlos Maldonado e Ronaldo Luiz Dias Cereda
*       Data: Abril/87
*       - Programa Principal -
*       Declaracao de variaveis, constantes simbolicas e iniciializacoes
        INTEGER CONT, J, LIMITE,NUMERO, VETRES(8)
        PARAMETER (LIMITE = 255)
        DATA VETRES/8*0/
*       Leitura e impressão do valor decimal
        WRITE(*,*) 'DIGITE O VALOR DECIMAL'
        READ(*,*) NUMERO
        WRITE(*,10) NUMERO

  10    FORMAT(1X,//,1X,'VALOR DECIMAL FORNECIDO: ',I3)
*       Verificacao do limite
```

```
      IF (NUMERO.GT.LIMITE) THEN
        WRITE(*,*) 'NUMERO MUITO GRANDE
      ELSE
*     Chamada da rotina para conversao e impressao dos resultados
      CONT = 0
      CALL CONV(NUMERO, VETRES, CONT)
      WRITE(*,20) (VETRES(J), J=CONT, 1, -1)
  20  FORMAT(1X, 'VALOR BINARIO CORRESPONDENTE: ', 8I2)
      ENDIF
      STOP
      END

*     Sub-rotina que faz a conversao de um numero decimal, maior
*     ou igual a zero, em um numero binario
*     Parametro de entrada
*          NUM: valor a ser convertido
*     Parametros de saida
*          VETRES: vetor que contem o resultado
*          PONT: numero de bits do resultado
      SUBROUTINE CONV(NUM, VETRES, PONT)
*     Declaracao dos parametros
      INTEGER NUM, PONT, VETRES(8)
*     Declaracao de variavel local
      INTEGER AUX
*     conversao
 100  IF (NUM.GE.2) THEN
*     inicio do enquanto
        PONT = PONT + 1
        AUX = NUM/2
        VETRES(PONT) = NUM - AUX*2
        NUM = AUX
      GOTO 100
*     fim do enquanto
      ENDIF
      PONT = PONT + 1
      VETRES(PONT) = NUM
      RETURN
      END
```

3.4.2 COBOL (COmmon Business Oriented Language)

A linguagem COBOL foi desenvolvida em 1959 pelo Departamento de Defesa dos EUA e fabricantes de computadores, foi o padrão para as aplicações comerciais e ainda hoje é muito utilizada. Seu desenvolvimento se deu de forma independente da máquina. O código é "English-like" e é excelente para a manipulação de arquivos.

Contribuições de COBOL para futuras linguagens:

- ***código mais legível***
- ***estrutura de dados heterogênea (record)***

3.4.3 ALGOL 60 (ALGorithmic Oriented Language)

Linguagem algébrica de origem européia, desenvolvida pelo comitê Internacional popular, destinada à resolução de problemas científicos. Influenciou o projeto de quase todas as linguagens projetadas a partir de 1960. Descrita em BNF (Backus-Naur Form), foi projetada independente da implementação, o que permite uma maior criatividade, porém de implementação mais difícil. É pouco usada em aplicações comerciais devido à ausência de facilidades de E/S na descrição e pelo pouco interesse de vendedores. Além disso, tornou-se padrão para a publicação de algoritmos.

Contribuições de ALGOL 60 para futuras linguagens:

- ***estrutura de blocos: habilidade de se criar blocos de comandos para o escopo de variáveis e extensão de influência de comandos de controle***
- ***comandos de controle estruturados: if-then-else e uso de uma condição geral para controle de iteração***
- ***recursividade: habilidade de um procedimento chamar a si próprio***

3.4.4 LISP (LISt Processing)

Linguagem funcional criada em 1960, por John McCartly do grupo de IA do MIT, para dar suporte à pesquisa em Inteligência Artificial, baseada inteiramente na Teoria Lambda Calculus. Foi inicialmente desenvolvida para o IBM 704. Desde o início sempre existiram muitos dialetos de LISP, tais como: MacLisp (MIT), InterLISP (DEC PDP-10), Spice LISP, Scheme (variante de InterLISP criada em 1970 por Greald Sussman e Guy Steele), etc. Em 1981, surgiu o Common LISP como uma primeira tentativa de padronização. Os anos entre 1985 e 1990 foram os de maior prosperidade para a linguagem, devido ao grande avanço que a área de Inteligência Artificial (IA) vinha alcançando. Em 1992, o comitê X3J13 publicou uma nova padronização de Common LISP que agregava o dialeto Scheme.

Contribuição de LISP para futuras linguagens:

- ***é pioneira na idéia de computação simbólica ou não numérica***

Um programa LISP utiliza uma lista encadeada como estrutura de dados básica. Sendo uma linguagem funcional, ao invés de especificar operações como um conjunto seqüencial de comandos, invoca funções e composição de funções para especificar múltiplas ações. Utiliza a mesma estrutura para dados e programas. Abaixo apresentamos o primeiro programa LISP, HelloWorld, que imprime a mensagem "Hello World" na tela. Observamos que um programa LISP é executado por um interpretador que executa as funções, previamente conhecidas (i.e. carregadas), pedidas pelo usuário.

Exemplos de programas LISP

1. **Programa HelloWord**

```
➢ lisp -> Para carregar o interpretador lisp

{Predicado que imprime a mensagem Hello World na tela.}
(define (hello i)
          (cond ((eq i 2) (print "Hello World"))
                (T (print "Goodbye World"))
          )
)
No prompt do interpretador ao executarmos o programa obteríamos:
➢ (hello 2)
Hello World
```

A partir de agora apresentamos algumas características básicas da linguagem para entendermos os exemplos apresentados.

A operação de atribuição é feita com a função **setq**. Enquanto que os predicados (**eq** x y)**,** (**and** x y)**,** (**or** x y)**,** (**not** x), (**null** x), são respectivamente a ***comparação***, ***e ou lógicos***, ***negação*** e ***testa de nulidade***. O operador **cons** é usado para construir listas enquanto os operadores **car** e **cdr** obtém respectivamente head e tail da lista passada como parâmetro. Vejamos o exemplo abaixo:

```
Se L=nil então
```

```
(setq L (cons A (cons B (cons C nil)))) fornece L=(A B C);
e ainda
(car L) é A; (car (cdr L)) é B; (car (cdr (cdr L))) é C;
```

O operador **cond** é a execução condicional com a seguinte sintaxe:

```
(cond  alternativa1
       alternativa2
       .......
       alternativan
       (T default_expression)
)
Onde alternativai é da forma (predicadoi expressãoi)
```

Os operadores **defun** (**define** em Scheme) são utilizados para criar novas funções:

```
(defun function_name(arguments) expression)
(define (function_name arguments) expression)
```

2. Programa Interseção de Dois Conjuntos

```
{Predicado que testa se dois conjuntos tem algum elemento em
comum. Retorna nil se os dois conjuntos são disjuntos.}
(defun INTERSECTP (A B)
  (cond   ((null A) NIL)
          ((member (CAR A) B))
          (T (INTERSECTP (CDR A) B)))
)
```

3.4.5 BASIC (Beginners All-purpose Symbolic Instruction Code)

A linguagem BASIC, desenvolvida em meados dos anos 60 por John Kemeny e Thomas Kurtz no Dartmouth College, teve como objetivo ensinar alunos de graduação a usarem um ambiente interativo de programação, através de uma LP de fácil aprendizado. Com o surgimento dos microcomputadores de baixo custo, no início dos anos 70, o BASIC tornou-se muito popular, embora não tenha contribuído muito tecnologicamente.

Contribuições de Basic para futuras linguagens:

- ***uma das primeiras LPs a prover um ambiente de programação interativo como parte da linguagem***
- ***execução interpretativa de programas***

Exemplo de programa Basic

```
010 REM ANAGRAMA PARA QUATRO LETRAS
020 PRINT "FORNECA QUATRO LETRAS QUAISQUER:"
030 PRINT
040 INPUT L$(1), L$(2), L$(3), L$(4)
050 PRINT
060 FOR I1 = 1 TO 4
070       FOR I2 = 1 TO 4
080             IF I2 = I1 THEN 150
090             FOR I3 = 1 TO 4
100                IF I3 = I1 THEN 140
110                IF I3 = I2 THEN 140
120                LET I4 = 10 - (I1 + I2 + I3)
130                PRINT L$(I1); L$(I2); L$(I3); L$(I4)
140            NEXT I3
150      NEXT I2
160 NEXT I1
170 END
RUN
```

3.4.6 PL/I (Programming Language I)

Desenvolvida em meados dos anos 60 pela IBM com o objetivo de incorporar características das LPs existentes numa única LP de propósito geral. Assim PL/I inclui:

- estrutura de bloco, de controle e recursividade do ALGOL 60;
- subprogramas e E/S formatadas do FORTRAN;
- manipulação de arquivos e registros do COBOL;
- alocação dinâmica de memória e estruturas encadeadas do LISP;
- operações de arrays do APL.

É uma linguagem difícil de aprender e implementar devido a sua grande complexidade. Além disso, faz uso de defaults.

Contribuições de PL/I para futuras linguagens:

- ***tratamento de interrupção - execução de procedimentos específicos quando uma condição excepcional ocorre***
- ***concorrência - especificação de tarefas serem executadas de forma concorrente***

3.4.7 SIMULA 67

Linguagem baseada em Algol 60, criada no início dos anos 60 por Ole Johan Dahl e Kristan Nygaard, na Noruega. É destinada à descrição de sistemas e programação de simulações.

Contribuição de SIMULA 67 para futuras linguagens:

- ***conceito de classe: Uma classe é um encapsulamento de dados e procedimentos que podem ser instanciados em um conjunto de objetos. É o conceito predecessor ao tipo abstrato de dados (ADA e Módula 2) e das classes das linguagens orientadas a objeto (Smalltalk, C++, Java , Eiffel)***

Exemplo de programa Simula 67:

Classe para manipulação de números complexos.

```
class complex(x, y); real x, y;
begin
ref(complex) procedure add (c); ref(complex) c;
  if c =/= none then add :- new complex(x + c.x, y + c.y);
ref(complex) procedure sub (c); ref(complex) c;
  if c =/= none then sub :- new complex(x - c.x, y - c.y);
ref(complex) procedure mult (c); ref(complex) c;
  if c =/= none then mult :- new complex(x * c.x - y * c.y,
                                         x * c.y + y * c.x);
ref(complex) procedure div (c); ref(complex) c;
begin real m;
      if c =/= none then begin m := c.modulus;
        if m ¬= 0.0 then div :- mult(c.conjugate).stretch(1/m);
      end;
end ***div***;
ref(complex) procedure conjugate; conjugate :- new complex(x, -y);
real procedure modulus; modulus := sqrt(x*x + y*y);
ref(complex) procedure stretch(k); real k;stretch:-newcomplex(k*x, k*y);
procedure write; begin
  outchar('(');outfix(x,3,12);outchar(',');outfix(y,3,12);outchar(')');
end ***write***
end ***complex***
```

3.4.8 ALGOL 68

É muito diferente do Algol 60. Linguagem de propósito geral que foi projetada para a comunicação de algoritmos, para sua execução eficiente em vários computadores e para ajudar seu ensino a estudantes. Porém é de difícil descrição, o que resultou em uma baixa popularidade.

Contribuição de ALGOL 68 para futuras linguagens:

- ***ortogonalidade: uma LP que é ortogonal tem um número de construtores básicos e um conjunto de regras para combiná-los relativamente pequeno (oposto a PL/I)***

Exemplo de programa ALGOL 68:

begin {Programa que lê três palavras de dez caracteres e as mostra em ordem alfabética }.

```
   [10]char s1, s2, s3;
   read((s1, s2, s3));
   if s1 < s2 then          { s1 < s2 e (s2 < s3 ou s2 > s3) }
     if s2 < s3 then
       print((s1, s2, s3))
     elif s1 < s3 then
       print((s1, s3, s2))
     else
       print((s3, s1, s2))
     fi
   elif s1 < s3 then        { s2 < s1 e s1 < s3 }
     print((s2, s1, s3))
   elif s2 < s3 then        { s2 < s1 e s1 > s3 e s2 < s3 }
     print((s2, s3, s1))
   else                     { s2 < s1 e s1 < s3 e s2 > s3 }
     print((s3, s2, s1))
   fi
end.
```

3.4.9 PROLOG (PROgramming in LOGic)

Linguagem desenvolvida em 1972 em Marseille na França. É destinada a aplicações de Inteligência Artificial e se baseia em lógica formal (Calculo de Predicados). É a LP do projeto japonês de quinta geração. Apesar de não existir nenhum padrão oficial, a versão desenvolvida pela universidade de Edinburgh (Arity Prolog) tem sido uma das mais usadas.

Um programa Prolog pode ser visto como um conjunto de Predicados (também chamadas de Regras de Conhecimento) que são utilizadas pelo motor de inferência do PROLOG para responder, **Yes** ou **No**, a perguntas feitas pelo usuário. Vejamos alguns exemplos de uma seção dentro de um interpretador Prolog

➢ **prolog -> Para carregar o interpretador prolog**

1. **Programa HelloWord**

```
/* Predicado que imprime a mensagem Hello World na tela. */
writeit :- write('Hello World'), nl.
```

No prompt do interpretador ao executarmos o programa obteríamos:

```
?- writeit.
Hello World
Yes.
```

2. **Programa Verificação de Data**

```
/* O programa abaixo retornará Yes a qualquer pergunta sobre a validade
de uma data dada. */

?- date(1964, 31, 12);
Yes
?- date(1999, 29, 2);
No

Regras de conhecimento previamente carregadas pelo interpretador,
através do comando abaixo:
?- date('pgm.prolog');

date(Year, Month, Day) :-
   Month > 0,  Month < 13,
   mdays(Year, Month, Ndays),
   Day > 0, Day =< Ndays.
mdays(_, 1, 31).
mdays(Y, 2, 28):- not leap(Y).
mdays(Y, 2, 29):- leap(Y).
mdays(_, 3, 31).
mdays(_, 4, 30).
mdays(_, 5, 31).
mdays(_, 6, 30).
mdays(_, 7, 31).
mdays(_, 8, 31).
mdays(_, 9, 30).
mdays(_, 10, 31).
mdays(_, 11, 30).
mdays(_, 12, 31).
leap(Y) :- divides(4, Y),
           not divides(100, Y).
leap(Y) :- divides(400, Y).
divides(Div, K) :-
   0 is K mod Div.
```

3.4.10 PASCAL

Desenvolvida por Niklaus Wirth em 1969, como uma alternativa para suplantar algumas deficiências encontradas na linguagem Algol. O nome Pascal foi escolhido em homenagem ao matemático Blaise Pascal que inventou a primeira máquina de calcular em 1642. O Pascal é uma linguagem de fácil aprendizado e implementação, suporta programação estruturada e é adequada para o ensino de programação. Em meados dos anos 80 também passou a ser usada para a programação em microcomputadores. Influenciou praticamente todas as linguagens mais recentes, especialmente Ada que herdou muitos dos seus conceitos.

Em 1968, Wirth começou a trabalhar em uma linguagem que seria a sucessora de ALGOL 60. Apesar de ALGOL ser a primeira linguagem a ter uma definição formal, a sua implementação se mostrou ser bastante complexa e muitos dos seus mecanismos, apesar de elegantes, não conseguiram ser implementados satisfatoriamente. Além disso, ALGOL não possuía nenhuma operação de entrada/saída por serem consideradas dependentes de máquina.

O objetivo de Wirth ao projetar esta nova linguagem era o de produzir uma linguagem que pudesse ser compilada em um passo por um interpretador que ficou conhecido como ***P-code Interpreter.*** A tarefa do compilador Pascal é traduzir o código fonte para uma linguagem de máquina hipotética (P-code) de uma arquitetura de zero endereços, também chamada de stack architecture. Dessa forma um **interpretador P-code** pode executar com relativa eficiência em qualquer plataforma, além de permitir um rápido transporte do compilador para outros sistemas computacionais. Uma vez escrito o código do compilador Pascal na própria linguagem Pascal, tudo o que precisa ser feito para transportar o compilador para outra máquina é

rescrever o **interpretador P-code** para esta nova máquina. Apesar dessa característica interessante a maioria dos compiladores hoje em dia prefere gerar código nativo da máquina alvo por razões de eficiência.

Apesar do rápido crescimento da linguagem desde1969, somente em 1984 o Pascal foi padronizado pela American National Standards Institute (ANSI) no comitê ANSI X3.97.

Contribuições de Pascal para futuras linguagens:

- ***estruturas de controle flexíveis***
- ***tipos definidos pelo usuário***
- ***arquivos***
- ***records***
- ***conjuntos***

Exemplos de programas em Pascal: ***HelloWorld, BuscaMaior e BuscaMenor elemento, SomaElemsVet .***

1. **Programa HelloWord**

```
(* Imprime a mensagem HelloWorld na tela. *)
PROGRAM HelloWorld;
VAR (* Variáveis Globais *)

PROCEDURE printmsg;
BEGIN
  Writeln('Hello World');
END; (* pesqMaior *)

BEGIN (* bloco principal *)
  printit;
END.
```

2. **Programa BuscaMaior**

```
(* Pesquisa o maior elemento de um vetor de n elementos *)
PROGRAM BuscaMaior(input, output);
CONST NELEMS = 100;
VAR (* Variaveis Globais *)
  n, i: 1..NELEMS;
  x: ARRAY [1..NELEMS] OF real;
  maior: real;

PROCEDURE pesqMaior;
(* Procura o maior elemento do vetor *)
BEGIN
  maior := x[1];
  FOR i := 2 TO n DO
    IF x[i] > maior THEN maior := x[i];
END; (* pesqMaior *)
```

```
VAR (* Variaveis do Main *)
BEGIN (* bloco principal *)
  write('Quantos sao os numeros? [ no msximo 100 ]');
  readln(n);
  FOR i := 1 TO n DO
  BEGIN
    write('x[', i:3, ']= ? ');
    readln(x[i]);
  END;
  pesqMaior;
  writeln;
  writeln('O maior elemento da sequencia eh: ',maior:4:1);
END.
```

3. Programa BuscaMenor

```
(* Pesquisa o menor elemento de um vetor de n elementos *)
PROGRAM BuscaMenor(input, output);
CONST NELEMS = 100;
TYPE  TVET = ARRAY [1..NELEMS] OF real;
VAR (* Variaveis Globais *)

FUNCTION pesqMenor( vet : TVET; nelems: integer): real;
(* Procura o menor elemento do vetor *)
VAR i: integer; menor: real;
BEGIN
  menor := vet[1];
  FOR i := 2 TO nelems DO
    IF vet[i] < menor THEN
       menor := vet[i];
  pesqMenor := menor;
END; (* pesqMenor *)

VAR (* Variaveis do Main *)
  n, i: 1..NELEMS;
  x: TVET;
  menor: real;
BEGIN (* bloco principal *)
  write('Quantos sao os numeros? [ no msximo 100 ]');
  readln(n);
  FOR i := 1 TO n DO
  BEGIN
    write('x[', i:3, ']= ? ');
    readln(x[i]);
  END;
  menor := PesqMenor(x, n);
  writeln;
  writeln('O menor elemento da sequencia eh: ',menor:4:1);
END.
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) pelos programas BuscaMaior e BuscaMenor no pior e no melhor caso ?

4. Programa Pesquisa Seqüencial

```
(* Pesquisa Seqüencial de um elemento em um vetor. Retorna o 1 quando o
elemento pesquisado for encontrado e -1 caso contrario. O parâmetro
indPesq, passado por referência, contém o índice do elemento pesquisado
*)
FUNCTION pesqSeq( chave: real; vet : TVET; nelems: integer;
                  var indPesq ) : integer;

(* Procura o elemento chave no vetor *)
VAR i: integer;
BEGIN
  indPesq := -1;
  pesqSeq := -1;
  FOR i := 1 TO nelems DO
    IF chave = vet[i] THEN
    Begin
       indPesq := i; pesqSeq := 1; Exit;
    End;
END; (* pesqSeq *)

VAR (* Variaveis do Main *)
  n, i: 1..NELEMS;  indPesq: integer;
  x: TVET;  chv: real;
BEGIN (* bloco principal *)
  write('Quantos sao os numeros? [ no msximo 100 ]');
  readln(n);
  FOR i := 1 TO n DO
  BEGIN
    write('x[', i:3, ']= ? ');
    readln(x[i]);
  END;
  write('Informe o elemento a pesquisar ? '); readln(chv)
  IF( pesqSeq(chv, x, n, indPesq) = 1 THEN
    writeln('Indice do elemento pesquisado e: ',indPesq)
  ELSE
    writeln('Elemento pesquisado não existe no vetor ');
END.
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa no pior e no melhor caso ?

5. Programa Soma de Elementos de um Vetor

```
(* Pesquisa o maior elemento de um vetor de n elementos *)
PROGRAM SomaElemensVet(input, output);
CONST NELEMS  = 100;
      ARQ_ENT = 'entrada.dat';
TYPE  TVET = ARRAY [1..NELEMS] OF real;
VAR (* Variaveis Globais *)

FUNCTION sum( vet : TVET; nelems: integer): real;
(* soma os elementos do vetor *)
VAR i: integer; temp: real;
BEGIN
  temp := 0;
  FOR i := 1 TO nelems DO temp:= temp + vet[i];
  sum := temp;
END; (* pesqMenor *)
```

```
VAR (* Variaveis do Main *)
  n, i: 1..NELEMS;
  x: TVET;
  infile: text;   (* arquivo com os dados de entrada *)
BEGIN (* bloco principal *)
  reset(infile, ARQ_ENT);  (* abre arquivo de entrada *)
  while not eof(infile) do
  BEGIN
    readln(infile, n);
    FOR i := 1 TO n DO
    BEGIN
      read(infile, x[i]);
      write('x[', i:3, ']= ? ', x[i]:10:2);
    END;
    writeln;
    writeln('Soma dos elementos da sequencia eh: ', sum(x,n):6:4);
    readln(infile);
  END;
END.
```

Exercício: Qual deve ser formato do arquivo de entrada e o formato da saída impressa por este programa ?

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa no pior e no melhor caso ?

3.4.11 C

A linguagem C foi criada por Dennis Ritchie e Ken Thompsom, em 1972, no centro de Pesquisas Bell Laboratories da AT&T. Tem seu estilo derivado de Algol e Pascal, e herdou algumas construções de PL/I. Sua primeira utilização importante foi a codificação do Kernel do Sistema Operacional UNIX, que até então era escrito em Assembly, para a máquina PDP-11 da DEC com 24K de memória principal. Em meados de 1970 o UNIX saiu do laboratório para ser liberado para as universidades.

Em 1978 Kernighan e Ritchie lançaram um livro contendo as definições básicas da linguagem C, que ficou conhecida como o **padrão C K&R**. O sucesso da linguagem foi tão grande que em 1980 já existiam várias versões de compiladores C oferecidas por várias empresas, não sendo mais restritas apenas ao ambiente UNIX, e também compatíveis com vários outros sistemas operacionais. Em 1983 o American National Standards Institute (ANSI) estabeleceu um comitê cujo objetivo foi produzir "***uma definição da linguagem C sem ambigüidades e independente de máquina".*** O resultado deste trabalho foi a publicação do **padrão ANSI-C.**

O C é uma linguagem de propósito geral, sendo adequada à programação estruturada. No entanto é mais utilizada para escrever compiladores, analisadores léxicos, bancos de dados, editores de texto, etc. O C pertence a uma família de linguagens cujas características são: **portabilidade**, **modularidade**, **compilação separada**, **recursos de baixo nível**, **geração de código eficiente**, **confiabilidade**, **regularidade**, **simplicidade** e **facilidade de uso.**

Exemplos de programas C: Contagem de Linhas; Métodos de Busca: Pesquisa Sequencial e Pesquisa Binária; , Métodos de Ordenação: Bolha e Shellsort; Soma de Elementos de um Vetor.

1. Programa HelloWord

```
/* Imprime a mensagem HelloWorld na tela. */
#include <stdio.h>

void main(void) {
   void printit(void);

    printit();
}
void printit(void) {
    printf("\nHello World");
}
```

2. Programa Contagem de Linhas

```
/* Contagem do numero de linhas na entrada padrão */
#include <stdio.h>

void main(void) {
   int c, nl=0;
   while( (c=getchar()) != EOF )
      if ( c == '\n' ) ++nl;
    printf("%d\n", nl);
}
```

Métodos de Busca: ***Pesquisa Seqüencial e Pesquisa Binária***

3. Programa Pesquisa Seqüencial e Binária

```
/* Pesquisa Seqüencial de um elemento em um vetor e pesquisa Binária de
um elemento em um vetor supostamente ordenado. Retorna 1 quando o
elemento pesquisado for encontrado e -1 caso contrario. O paramento
indPesq contem índice do elemento pesquisado */

#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/* Declaração de Constantes e Tipos do usuário */
#define NELEMS 100
typedef enum { CHEFE, ENG, TEC } TCARGO;
typedef struct {  char nome[30];
                  TCARGO cargo;
                  float sal;
               } TTAB;

int PesqSeq(char chave[], TTAB pTab[], int nRegs, int *indPesq)
{
  int i;
  for( *indPesq=-1,i=0; i<nRegs; i++ ) {
    if( strcmp(chave, pTab[i].nome) == 0 ) {
      *indPesq = i; return 1;
    }
  }
  return -1;
} /* Fim de PesqSeq */
```

```
int PesqBin(char chave[], TTAB pTab[], int nRegs, int *indPesq)
{
  int comp, inicio=0, fim=nRegs-1, meio;
  if(nRegs == 0) {
   indPesq = 0; return -1;

  while( inicio <= fim ) {
    meio = (inicio+fim)/2;
    comp = strcmp(chave, pTab[meio].nome);  /* ordem lexicográfica */
    if( comp == 0 ) {     /* Elemento encontrado na posição "meio" */
      *indPesq = meio;
       return 1;
    } else if( comp < 0 )
      fim=meio-1;
    else
      inicio=meio+1;
  }
  if( comp > 0 )          /* Elemento não encontrado */
    *indPesq = meio + 1; /* Posição a ser inserido o elemento */
  else
    *indPesq = meio;     /* Posição a ser inserido o elemento */
  return -1;

} /* Fim de PesqBin */


void main(void) {
int n;
float x[NELEMS], chave;

  printf("\nQuantos sao os numeros? [ no msximo 100 ]");
  scanf("%d", &n);
  for( i=0; i<n; ++i ) {
    printf("x[%d] = ?", i);
    scanf("%f", &x[i]);
  }
  printf("\nInforme o elemento a pesquisar ? "); scanf("%f", &chave);
  if( PesqSeq(chv, x, n, &indPesq) )
    printf("\nIndice do elemento pesquisado e: %d", indPesq);
  else
    printf("\nElemento pesquisado não existe no vetor !");

  /* Ordena vetor para pesquisar o elemento chave */
  Bolha(x, n);

  printf("\nInforme o elemento a pesquisar ? "); scanf("%f", &chave);
  if( PesqBin(chv, x, n, &indPesq) )
    printf("\nIndice do elemento pesquisado e: %d", indPesq);
  else
    printf("\nElemento pesquisado não existe no vetor !");

} /* Fim de main */
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa para a PesqSeq e para a PesqBin, no pior e no melhor caso ?

Métodos de Ordenação: ***Bolha, ShellSort***

4. Programa Ordenação por Trocas Sucessivas

```
/* Ordenação por Trocas sucessivas - Método da Bolha
Para ordenar um vetor podemos percorrer o vetor da primeira a ultima
posição, i=0 até n-1, e comparar sucessivamente elementos v[j] e v[j+1],
j=i até n-1, e fazer a inversão sempre que necessário */

void troca (float *a, float *b ) {
  float temp =  * a ;
  *a = * b;
  *b = * a;
}

void Bolha ( float v[ ], int n )
{
  int i, j;
  float temp;
  for( i=0; i < n-1; i++ )
    for( j=i; j >= 0 : --j )
      if( v[j] > v[j+1] )
        troca(&v[j], &v[j+1]);
} /* Fim de Bolha */
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa no pior e no melhor caso ?

5. Programa Ordenação Shell

```
/* Ordenação Shell - D. L. Shell em 1959
Idéia básica deste algoritmo, é  em estágios iniciais, elementos
distantes são comparados, ao invés de elementos adjacentes, como no
método anterior (Bolha). Isto tende  a eliminar a grande quantidade de
desordem rapidamente, de modo que estágios posteriores  tenham menos
trabalho a fazer. O intervalo entre elementos  comprados é gradualmente
decrementado até um, em cujo ponto a ordenação se torna efetivamente um
método de troca adjacente. */

void ShellSort( float v[ ], int n )
{
  int i, j, temp;
  for( inter=n/2; inter > 0; inter/= 2 )
    for( i=inter; i < n : ++i )
      for( j=i-inter; j>=0 && v[j+inter]; j-=inter }
        troca(&v[j], &v[j + inter]);
} /* Fim de ShellSort */
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa no pior e no melhor caso ?

6. Programa Soma de Elementos de um Vetor

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

/* Declaração de Constantes e Tipos do usuário */
#define NELEMS  100
#define ARQ_ENT "entrada.dat"

float Sum(real vet[], int nelems) {
  int i; float temp;
  for( i=temp=0; i<nelems; ++i )
    temp += vet[i];
  return temp;
} /* Fim de Sum */

void main(void) {
int n;
float x[NELEMS];
FILE *infile;

  if( (infile=fopen(ARQ_ENT, "r") == NULL ) {
    printf("\n Erro ao abrir arquivo %s", ARQ_ENT)';
    getchar();
    exit(1);
  }
  while ( 1 ) {
    fscanf(infile, "%d", &n);
    if ( feof(infile) )
      break;
    for( i=1; i<n; ++i ) {
      fscanf(infile, "%f", &x[i]);
      printf("x[%d]= %10.2f", i, x[i]);
    }
    printf("\nSoma dos elementos da sequencia eh: %6.4f", Sum(x,n));
  }
  if( fclose(infile) == EOF ) {
    printf("\n Erro ao fechar arquivo %s", ARQ_ENT)';
    getchar();
    exit(1);
  }
} /* Fim de main */
```

Exercício: Modifique o programa anterior para escrever a saída em um arquivo texto ao invés de sair na tela.

3.4.12 SMALLTALK

Criada por Alan Kay da Xerox - Palo Alto no início dos anos 70. Apresenta um ambiente de programação com menus pop-up, windows e mouse (modelo para Apple Macintosh). Segue o modelo orientado a objetos, possuindo o conceito de classe do SIMULA 67 mais encapsulamento, herança e instanciação.

Contribuições de Smalltalk para futuras linguagens:

- ***primeira LP a utilizar o paradigma de programação interativa***
- ***introduz o conceito de LP extensível***

Exemplo de programa Smalltalk:

Classe que representa o objeto Linha.

```
Class Line
|startPoint endPoint|
[
from: sPoint to: ePoint
   startPoint <- sPoint.
   endPoint <- ePoint
|
+ offset
   ^ Line new; from: (startPoint + offset)
     to: (endPoint + offset)
|
drawWith: APen
   aPen up.
   aPen goTo: startPoint.
   aPen down.
   aPen goTo: endPoint
]
```

3.4.13 MÓDULA 2

Criada por Niklaus Wirth no final dos anos 70, é uma linguagem de propósito geral, baseada em melhorias no Pascal. É boa para projetos de desenvolvimento de software de grande porte. Além disso foi usada para ensinar programação. Módula 2 elaborou Pascal em:

- módulos podem ser usados para implementar **TAD (Tipos Abstratos de Dados)**
- todas as estruturas de controle têm uma palavra-chave de terminação
- co-rotinas - execução intercalada
- tipos de procedimentos

Exemplo de programa Módula 2:

Programa que lista todas as possíveis permutações de n objetos distintos a[1] ... a[n].

```
MODULE Permute;
 FROM InOut IMPORT Read, Write, WriteLn;
 VAR n: CARDINAL; ch: CHAR;
     a: ARRAY[1..20] OF CHAR;
 PROCEDURE output;
  VAR i: CARDINAL;
 BEGIN
  FOR i := 1 TO n DO Write(a[i])END;
  WriteLn;
 END output;

 PROCEDURE permute(k: CARDINAL);
  VAR i: CARDINAL; t: CHAR;
 BEGIN
  IF k = 1 THEN output
  ELSE permute(k-1);
   FOR i := 1 TO k-1 DO
    t := a[i]; a[i] := a[k]; a[k] := t;
    permute(k-1);
    t := a[i]; a[i] := a[k]; a[k] := t;
   END
  END
 END permute
```

```
BEGIN Write(""); n := 0; Read(ch);
 WHILE ch  "" DO
  n := n + 1; a[n] := ch; Write(ch); Read(ch);
 END;
 Writeln; permute(n)
END Permute.
```

3.4.14 C++

Assim como Pascal tem seu desenvolvimento devido a pessoa de Niklaus Wirth, também a linguagem C++ esta associada a Bjarne Stroustrup, que em 1981, desenvolveu o C++ como uma extensão da linguagem C. Stroustrup conseguiu adicionar poderosos conceitos de programação orientada a objetos (OOP - Object Oriented Programming), tais como; **Herança, Polimorfismo e Encapsulamento**, mantendo a eficiência de C.

Em 1982, Stroustrup, então pesquisador da AT&T Bell Laboratories, criou extensões a linguagem C, inspirado na sua experiência de programação com a linguagem SIMULA durante a sua tese de doutorado. À estas extensões ele chamou de "C com classes". Em 1989, foi criado o comitê ANSI X3J16 para definir um padrão para o C++ que foi publicado em 1994.

Contribuições de C++ para futuras linguagens:

- ***Criação do Conceito de Classes abstratas***
- ***Possibilidade de uso de Late-Binding ao invés de Early-Binding***
- ***Definição de templates***

Exemplo de programa C++:

1. **Programa HelloWord**

```
#include <iostream.h>  // C++ IO streams análogo ao stdio.h

class messages {
  public:
    void printit(void) { cout << "\nHello World" << endl; };
}
void main(void) {
   messages x;
    x.printit();
}
```

2. **Programa Soma de Elementos de um Vetor**

```
#include<fstream.h>
#include<stdlib.h>

/* Declaração de Constantes e Tipos do usuário */
#define NELEMS  100
#define ARQ_ENT "entrada.dat"

class DataStore{
  public:
    int nelems;
    int vet[NELEMS];
    float sum(void){ int i; float temp;
                     for( i=temp=0; i<nelems; ++i )
                       temp += vet[i];
```

```
                        return temp; }
}

void main(void) {
DataStore x;
int i;
ifstream infile(ARQ_ENT);

  if( !infile ) {
    cout << endl << "Erro ao abrir arquivo " << ARQ_ENT;
    cin.get();
    exit(1);
  }
  while ( 1 ) {
    infile >> x.nelems;
    if ( infile.feof() )
      break;
    for( i=0; i<x.nelems; ++i ) {
      infile >> x.vet[i];
      cout << "x.vet[" << i << "]=" << x.vet[i];
    }
    cout << endl << "Soma dos elementos da sequencia eh" << x.Sum();
  }
  infile.close();

} /* Fim de main */
```

Exercício: Modifique o programa anterior de forma a colocar as variáveis nelems e vet da classe DataStore como privadas ao invés de públicas para reforçar o encapsulamento.

Exercício: Modifique o programa do exercício anterior para escrever a saída em um arquivo texto ao invés de sair na tela.

3.4.15 Java

Java é uma linguagem projetada para programação dentro da Web. Derivada da Linguagem OAK, que foi especialmente projetada para programar aparelhos eletro-eletrônicos por James Gosling da Sun em 1990. Em 1993 passa a se chamar Java e sua principal característica é sua independência de plataforma. O HotJava é o browser da Sun criado especialmente para dar suporte a execução de código Java na Web.

As principais características de Java são: Simplicidade; Linguagem totalmente Object Oriented, recursos para a computação distribuída, interpretada, robusta, segura, independente de plataforma (architecture-neutral), portável, alta performance, multithreaded e dinâmica..

Exemplos de programa Java

1. **Programa HelloWord**

```
import java.*;

class JHelloWorld {

  public static void main( String argv[] ) {
    System.out.println("\nHello World");
    System.exit(1);
  }

} // Fim class JHelloWorld
```

Métodos de Ordenação: ***JSelectionSort e JQuickSort***

2. Programa JSelectionSort

```
/* Ordenação SelectionSort
Dado um vetor com n elementos selecione o menor item do vetor e a seguir
troque-o com o item que está na primeira posição do vetor. Repetindo-se
esta operação com n-1 itens restantes, depois com os n-2 restantes e
assim sucessivamente até que reste apenas um elemento. A seguir
apresentamos duas versões para esse algoritmo sendo uma recursiva. */

import java.*;

class JSelectionSort {
  public static void main( String argv[] ) {
    // Critica entrada de dados
    if( argv.length < 2 ) {
      System.out.println("\nSintaxe: java SelectSort {R|L} n1 n2 ...");
      System.exit(1);
    }
    // Aloca vetor com os parâmetros
    int [] vet = new int [argv.length-1];

    for( int i=0; i<vet.length; ++i ) {
      vet[i] = Integer.parseInt(argv[i+1]);
    }
    // Ordenar vetor ...
    JSelectionSort s = new JSelectionSort();
    System.out.println( "\nVet antes do sort" );
    for( int i=0; i<vet.length; ++i ) {
      System.out.println( "vet[" +i+"]= " + vet[i]);
    }
    if ( argv[0].trim().toUpperCase().compareTo("R")== 1 )
      s.sortR(vet, 0);
    else
      s.sortL(vet);

    System.out.println( "\nVet apos o sort " + argv[0] );
    for( int i=0; i<vet.length; ++i ) {
      System.out.println( "vet[" +i+"]= " + vet[i]);
    }
  }
  // Versão Recursiva
  void sortR( int vet[], int ini ) {
    if ( ini == vet.length-1 )
      return;
    } else {
      // Procura menor do vetor entre ini e vet.length
      int menor= vet[ini], indMenor= ini;
      for( int i=ini+1; i<vet.length; ++i ){
        if( vet[i] < menor ) {
          indMenor = i;  menor=vet[i];
        }
      }
      // Troca menor para a posicao ini
      if ( indMenor != ini ) {
          int temp = vet[ini]; vet[ini] = vet[indMenor];
          vet[indMenor] = temp;
      }
      // Recursão ...
      sortR( vet, ++ini);
    }
```

```
  }

  // Versão Não Recursiva
  void sortL( int vet[] ) {
    // Procura menor do vetor entre ini e vet.length
    for( int ini=0; ini<vet.length; ++ini ) {
      int menor= vet[ini], indMenor= ini;
      for( int i=ini+1; i<vet.length; ++i ){
        if( vet[i] < menor ) {
          indMenor = i; menor=vet[i];
        }
      }
      // Troca menor para a posicao ini
      if ( indMenor != ini ) {
        int temp = vet[ini];
        vet[ini] = vet[indMenor]; vet[indMenor] = temp;
      }
    }
  }
} // Fim class JSelectionSort
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa, para as versões sortR (recursiva) e sortL (não recursiva), no pior e no melhor caso ?

3. Programa JQuickSort

```
/* Ordenação QuickSort - C. A. Hoare em 1962
Dado um vetor de elementos escolher um elemento pivot para fazer uma
partição no vetor dividindo-o em dois subconjuntos - um subconjunto com
aqueles elementos que são menores que o pivot e o outro subconjunto com
os elementos que são maiores que o pivot. O mesmo processo é então
aplicado recursivamente a cada um dos dois subconjuntos. Quando um
subconjunto tiver menos de dois elementos, ele não precisa de outra
ordenação e isso encerrará a recursão */

import java.*;

class JQuickSort {

  public static void main( String argv[] ) {
    // Critica entrada de dados
    if( argv.length < 2 ) {
      System.out.println("\n Sintaxe: java QuickSort n1 n2 ...");
      System.exit(1);
    }
    // Aloca vetor com os parametros
    int [] vet = new int [argv.length];

    for( int i=0; i<vet.length; ++i ) {
       vet[i] = Integer.parseInt(argv[i]);
    }
    // Ordena vetor ...
    JQuickSort s = new JQuickSort();
    System.out.println( "\nVet antes do sort" );
    for( int i=0; i<vet.length; ++i ) {
       System.out.println( "vet[" +i+"]= " + vet[i]);
    }
    // Chama método de sort ...
    s.quickSort(vet, 0, vet.length-1);
    System.out.println( "\nVet apos o sort " + argv[0] );
    for( int i=0; i<vet.length; ++i ) {
       System.out.println( "vet[" +i+"]= " + vet[i]);
```

```
        }
    } // Fim do main

    // Metodos publicos
    void quickSort( int vet[], int esq, int dir ) {
       // Procura menor do vetor entre esq e vet.length
       int i=esq, j=dir;
       int pivot=vet[(i+j)/2];
       do {
            while(vet[i]<pivot && i<dir) ++i;
            while(pivot<vet[j] && j>esq) --j;
            if( i<=j )
               troca(vet, i++, j--);
       } while ( i<=j );
       // Recursão ...
       if( esq<j ) quickSort(vet, esq, j);
       if( i<dir ) quickSort(vet, i, dir);
    } // Fim do quickSort

    // Métodos privados
    private void troca(int vet[], int i, int j ) {
       int temp = vet[i];
       vet[i] = vet[j];
       vet[j] = temp;
    }
} // Fim class JQuickSort
```

Exercício: Considerando-se que o número de dados de entrada seja um inteiro n, qualquer, diga quantas comparações serão feitas (em função de n) por este programa no pior e no melhor caso ?

Exercício: Implemente uma versão não recursiva do algoritmo QuickSort.

4 Conceitos Básicos em Linguagens de Programação

4.1 Processamento de Linguagens

Usuários distintos entendem um computador de forma distinta, de acordo com a interface definida pelo software que costumam utilizar. Nos casos extremos, é possível que dois usuários da mesma máquina não consigam sequer uma base comum para trocar impressões sobre a .máquina que ambos utilizam, simplesmente porque cada um conhece apenas uma face, ou interface, distinta ca máquina. definida pelo software e pelos periféricos que utiliza.

Para cada linguagem de programação L, a máquina pode ser vista como um sistema exclusivamente dedicado à execução de programas em L. Diferentes linguagens fazem com que aparentemente tenhamos diferentes **máquinas virtuais**, cuja implementação não interessa ao usuário, na maioria dos casos. Em princípio, pelo menos, para quem executa um programa escrito em Pascal não faz diferença se o hardware da máquina executa o código Pascal diretamente, ou se uma tradução é feita para o código que é finalmente executado (a linguagem da máquina), possivelmente em vários passos.

A **máquina virtual** de uma linguagem de programação L pode sempre ser vista como a implementação de uma interface entre a máquina e o usuário, interface essa definida através de L. Essa implementação pode ser, até certo ponto, estendida ou modificada pelo usuário: se um programador de Pascal dispõe das declarações de um tipo chamado **Complex**, e de um conjunto de rotinas suficientes para as ações que pretende executar com valores do novo tipo, esse tipo pode ser considerado como implementado pela nova máquina virtual, obtida por extensão da anterior: um Pascal com números complexos.

Na prática, a implementação de uma máquina virtual nunca é totalmente transparente (invisível): alguns aspectos da forma pela qual foi feita podem se tomar aparentes para seus usuários, através do tempo gasto para a execução, ou de mensagens provenientes de etapas intermediárias da implementação, aspectos esses que deveriam ser invisíveis.

Para implementar uma linguagem de programação diretamente numa máquina, esta deve dispor de circuitos que, para cada instrução da linguagem. se encarregam das seguintes ações:

1. busca na próxima instrução a ser executada;
2. análise da instrução, determinando a ação que deve ser executada, como devem sei obtidos os dados de entrada para a execução dessa ação, e a forma de tratamento dos seus resultados;
3. busca dos dados necessários;
4. execução da ação correspondente sobre esses dados
5. armazenamento, (ou outro tratamento adequado) dos resultados.

Mesmo para linguagens de máquina, esta implementação não precisa ser feita diretamente, e pode ser *feita através* de um passo intermediário através da execução do micro-programa associado a cada instrução. Neste caso, as instruções são simuladas por programas embutidos no próprio processador, em geral na unidade de controle (UC).

As duas formas básicas de implementação de uma linguagem de programação são a **compilação** e a **interpretação**, que são frequentemente usadas em de forma combinada.

4.2 Compilação x Interpretação

Um **Compilador** implementa urna linguagem fonte *traduzindo* programas escritos nessa linguagem para a linguagem objeto da máquina alvo, onde os programas irão ser executados. A vantagem é que o compilador precisa traduzir um comando apenas uma única vez, não importando quantas vezes ele será executado. Na prática o compilador é usado para gerar o código definitivo (eficiente) de um programa.

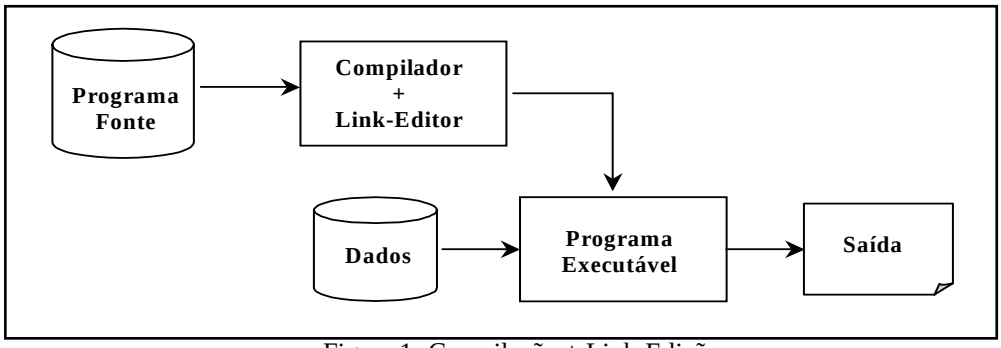
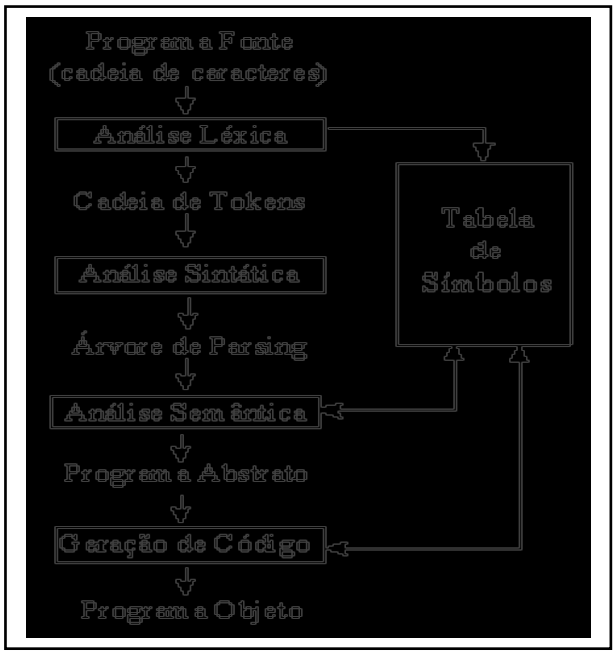


Figura 1- Compilação + Link-Edição

As fases principais da **Compilação :**



Um **interpretador** examina o programa *fonte,* e *simula* a execução de cada instrução ou comando de forma que o seu efeito seja reproduzido corretamente, à medida que essa execução se torna necessária. Essencialmente, o funcionamento de um interpretador pode ser descrito pelas ações (1) - (5) acima. De certa maneira, podemos considerar que o hardware de um computador, ao executar um programa armazenado em sua memória, faz a interpretação desse programa, instrução a instrução. A vantagem é que o interpretador não traduz comandos que podem não ser executados e pode relatar erros na linguagem original em cada ponto de execução. Na prática as linguagens interpretadas servem para a realização de uma prototipagem rápida.

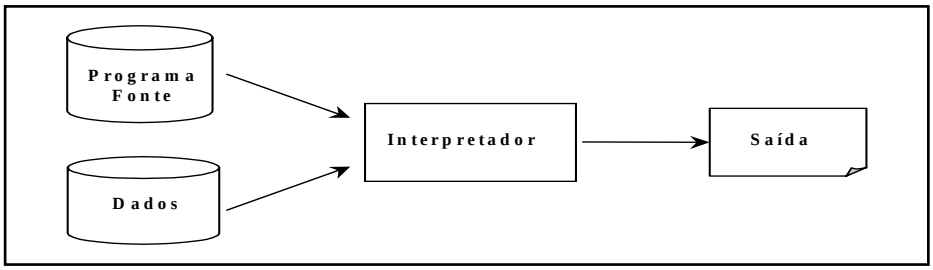


Figura 2- Interpretação Pura

Na prática, não existem compiladores ou interpretadores puros: cada **Compilador** ou **Interpretador** recebe o seu nome em função da forma de implementação que melhor o descreve.

Por exemplo, seria desnecessário que um **Compilador** traduzisse cada acesso a um arquivo em disco feito em um programa inserindo no programa objeto várias vezes o mesmo código necessário para obrigar o hardware do disco a executar a sequência de ações correspondentes. O que se faz na prática é interpretar essas instruções através de chamadas a rotinas, conhecidas como *serviços,* do sistema operacional (system calls), que se tornam responsáveis por essas ações.

Por outro lado muitos **Interpretadores** efetuam urna tradução do código fonte para uma representação interna ou código intermediário, cuja interpretação pode então ser feita com maior facilidade.

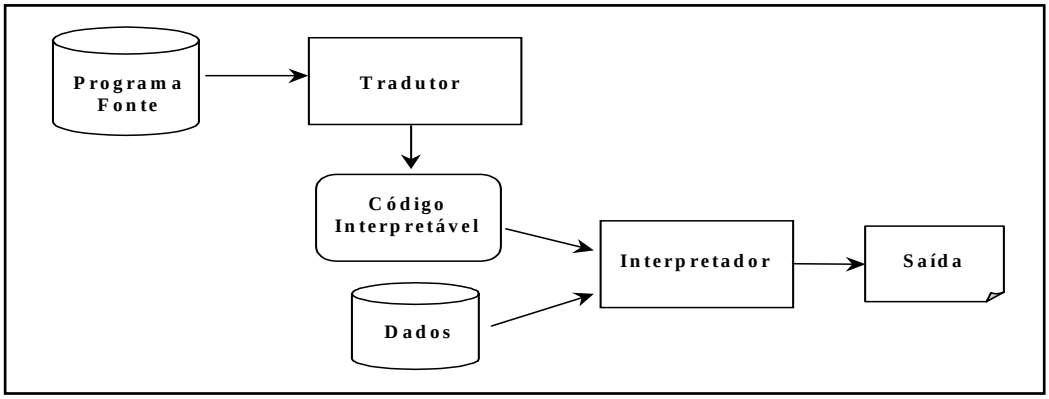


Figura 3- Interpretação com tradução prévia

Uma pequena confusão se pode fazer neste último caso: o programa responsável pela tradução acima mencionada é às vezes chamado de **Interpretador**, *às* vezes de **Compilador**. Quando se constrói um programa objeto executável a partir do código intermediário gerado, o programa responsável pela interpretação é incluído automaticamente, e a forma de implementação pede não ficar clara. Esta situação pode ser vista na figura 4.

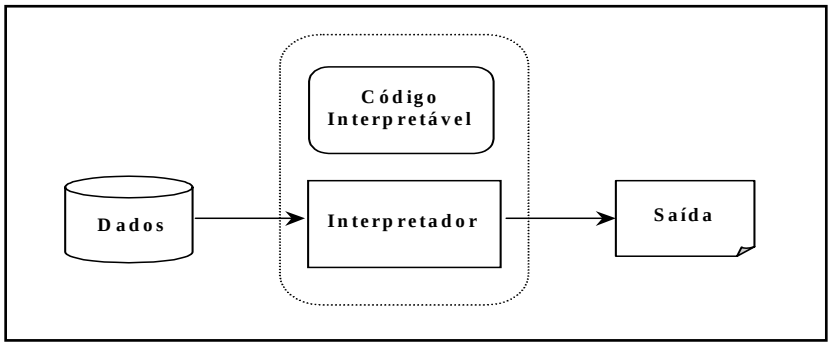


Figura 4- Compilação simulada por interpretação

Frequentemente, se nenhuma menção é feita, as diferenças entre **Interpretadores** e **Compiladores** não pode ser reconhecida externamente com facilidade. Entretanto, em geral **Interpretador** é mais lento, uma vez que alguns passos (essencialmente, os passos (1) e (2) acima) são repetidos cada vez que uma instrução deve ser executada; no caso do **Compilador**, isso não é necessário. Adicionalmente, é possível adaptar o código gerado pelo **Compilador** para cada comando de acordo com seu contexto, tirando proveito de diferenças que não podem ser tratadas com a mesma facilidade pelo **Interpretador**. Por outro lado, em geral o controle que o **Interpretadores** tem sobre o programa que está sendo executado é maior, e é possível detectar durante a execução situações de erro invisíveis (ou imprevisíveis) para o compilador durante a fase de tradução.

4.3 Conceito de Ligação ou Amarração

Durante o processo de programação, é necessário estabelecer uma correspondência entre os objetos em termos dos quais a especificação do programa foi construída, e objetos da linguagem de programação. Essa correspondência é feita através de nomes definidos pelo programador, que pela implementação devem corresponder a objetos concretos, ou reais, da máquina virtual, que de alguma forma vão posteriormente ser associados a posições de memória e/ou trechos de código durante a execução. A definição dessa correspondência é feita de acordo com regras especificadas para cada linguagem, e pode ser complicada, porque :

1 O **objeto "concreto"** associado ao nome pode não existir, ou pode estar invisível (isto é, existe, mas as regras da linguagem proíbem o acesso ao objeto) em certos trechos da execução do programa;

2 O **objeto pode ser "instanciado"** mais de uma vez, de forma que vários **objetos "concretos"** existem (ou seja, tem vida) simultaneamente, correspondendo ao mesmo nome declarado pelo programador, e é necessário que a instância correta seja acessada.

Fala-se em **ligação** ***(Binding)*** para definir essa correspondência entre nomes e objetos. Conforme o caso, a correspondência pode ser definida por ocasião da programação, da tradução (compilação) ou da execução. Diz-se que a correspondência é feita então, ***em tempo de programação, tradução (compilação + ligação - Early-Binding) ou execução (Late-Binding).***

Um objeto '1' de uma linguagem tem a ele associado o valor inteiro 1, e pode ser considerado pré-definido. Um objeto de nome ***"i"*** é provavelmente uma variável de tipo inteiro, e o nome "**i**" estará associado a uma ou mais posições da memória onde se pode armazenar um valor inteiro. Dependendo da linguagem, a situação varia: em FORTRAN, a alocação de espaço é feita estaticamente, e uma posição é escolhida para "**i**". durante a compilação, ficando definitivamente estabelecida a ligação entre o nome "**i**" e aquela posição de memória.

Em outras linguagens, como Pascal, essa ligação só pode ser feita dinamicamente, durante a execução do programa: como em Pascal podemos ter procedimentos recursivos, uma variável local a procedimento pode ter várias instâncias, cujo número e localização vão depender da forma da execução. Note, entretanto, que, para o programador, neste caso, o nome "**i**" sempre corresponde a um único objeto (a variável "i" da instância), e, durante a programação, as ações a serem executadas com qualquer instância da variável "i" são especificadas uma única vez.

4.4 Abstração de Dados e de Controle

Abstração é o processo de identificar as qualidades ou propriedades importantes do fenômeno sendo modelado. Usando o modelo abstrato pode-se concentrar unicamente nas qualidades ou propriedades relevantes e ignorar as irrelevantes. A título de exemplo vejamos o caso de uma pessoa que esteja aprendendo a dirigir. Nesse caso a pessoa precisa apenas se concentrar em alguns dos componentes do carro que a auxiliam a dirigir tais como: o pedal do acelerador, o pedal do freio, o volante e a embreagem. Ao passo que um engenheiro que esteja projetando um novo carro deve conhecer mais a fundo a relação entre pedais e motor, isto é, dizemos que os dois indivíduos trabalham com diferentes níveis de **Abstração** para realizar a sua tarefa.

Programas de Computador podem ser vistos como modelos de algum procedimento. Um programa pode ser visto como uma abstração da realidade. O processo de programação exige do programador uma tradução entre duas linguagens, a saber:

- ***A linguagem do problema****,* isto é, a linguagem em que as ações a programar e os objetos a que essas ações se referem estão descritos, em alguma forma de representação do problema que se busca resolver;

- **A *linguagem de programação****,* que oferece ao programador um conjunto de mecanismos de programação que foram julgados adequados pelo projetista da linguagem, de acordo com a sua finalidade prevista.

A tradução entre as duas linguagens pode ser mais simples ou mais complicada em função da correspondência que possa ser feita entre os mecanismos oferecidos pela linguagem de programação e os objetos e ações usados na descrição do problema. Normalmente, a correspondência mencionada acima não é completa, e toma-se necessário fazer a correspondência de um objeto ou ação da linguagem do problema a um conjunto de objetos ou sequência de ações da linguagem de programação.

Este processo peio qual os objetos ou ações compostas podem ser considerados como unidades é uma forma de **Abstraçâo***.* Diz-se que uma linguagem de programação tem facilidades para **Abstraçâo** se *é* possível definir objetos e ações compostos (abstratos } e tratá-los como se sempre tivessem feito parte da linguagem, fazendo referência a eles através de seus nomes.

As primeiras linguagens de programação, não reconheciam o papel crucial que a **Abstraçâo** desempenha na programação. No inicio da década de 50, a Linguagem Assembly (ou de Montagem) incorporou o uso de nomes simbólicos para representar códigos de operação de instrução e endereços de memória para a linguagem de máquina (o que melhorava a legibilidade e a manutenibilidade dos programas).

Subprogramas (e macros) também foram introduzidos pela linguagem de Assembly como um meio do programador nomear uma atividade descrita por um grupo de ações e considerá-la como uma ação única. São ferramentas úteis para a programação metódica, pois são mecanismos para construir abstrações. Hoje em dia, praticamente todas as linguagens oferecem mecanismos de **Abstraçâo de Procedimento**.

Outro mecanismo de **Abstraçâo** extremamente comum é o agrupamento de várias variáveis em uma única, corno por exemplo é feito pela declaração DIMENSI0N em FORTRAN ou pelo uso de tipos ARRAY em linguagens como Pascal ou AIgol-68. Em FORTRAN, uma declaração DIMENSION X(100) corresponde à declaração de 100 variáveis reais X(1), X(2), ..., X(100), que, de forma equivalente, pode ser entendida como a declaração de uma única variável X. Este mecanismo é um exemplo de **Abstraçâo de Dados.**

4.4.1 Exemplos de Abstração de Dados

Abaixo apresentamos alguns exemplos do uso da **Abstraçâo de Dados**, tomando como exemplo a linguagem C.

As definições de novos tipo permite ao programador uma melhor modelagem do problema dentro da aplicação, a seguir apresentamos algumas modelagens.

1. **Modelagem de Dados para Registro de Funcionário**

```
/* Definição de Tipos */
typedef  enum {   OK, ERRO } TRET;
typedef  enum {   CHEFE, ENG, TEC } TCARGO;
typedef  struct {   char      nome[30];
                    TCARGO   cargo;    float   sal; } TFUNC;
```

TRET pesqBin(char nome[ ], TFUNC pTab[ ], **int nRegs, int *pos** ***);***

2. **Tipo Abstrato de Dados Pilha - Comportamento LIFO**

/* Funções Utilitárias */
1. ***Cria*** – para criação do TAD Pilha;
2. ***Top*** – para consultar o elemento do início da Pilha;
3. ***Push*** – inserir elemento na Pilha;
4. ***Pop*** – remover elemento da Pilha;
5. ***Vazia e Cheia*** – testa se a Pilha esta vazia ou cheia;
6. ***Destroi*** – para remoção do TAD Pilha;

```
/* Definição de Tipos */
#define  MAXELEMS   100
typedef  int TPELEM;
typedef  struct  {
 short int      topo; ;                    /* topo da Pilha */
     TPELEM     vet[MAXELEMS];    /* Vetor que contem os elementos da Pilha */
} TPILHA;
```

Exemplo de uso:
```
TPILHA p;    Cria( &p );
Push(&p, 10); Push(&p, 15);
Destroi( &p );
```

3. **Tipo Abstrato de Dados Fila - Comportamento FIFO**

/* Funções Utilitárias */
7. ***Cria*** – para criação do TAD Fila;
8. ***Front*** – para obter o elemento do início da Fila;
9. ***Insere*** – inserir elemento do inicio da Fila;
10. ***Remove*** – remover elemento no final da Fila;
11. ***Vazia e Cheia*** – testa se a Fila esta vazia ou cheia;
12. ***Destroi*** – para remoção do TAD Fila;

```
/* Definição de Tipos */
#define MAXELEMS   100
typedef   struct  {
 int           head, tail;          /* Indices em vet do inicio e final da Fila */
    int          nelems;              /* Numero de Elementos usados em vet */
    TPELEM   vet[MAXELEMS];    /* Vetor que contem os elementos da Fila */
} TFILA;
```

Abaixo redefinimos ***TAD FILA*** para trabalhar com elementos ***complexos***

```
typedef  struct {   float   real;  float   imag; } COMPLEX;
```

typedef COMPLEX TPELEM;	***int Insere(TFILA fila, TPELEM elem);***
TPELEM Front(TFILA fila);	***TPELEM*** *Remove(TFILA fila);*

Exemplo de uso:

```
TFILA f;  COMPLEX  x = { 0, 0 }, y;
Cria( &f );
Insere(&f, x); Remove(&f, &y);
printf("\n Elemento removido y=(%f, %f)", y.real, y.imag);
Destroi( &f );
```

4. <u>Modelagem de um Hotel</u>

```
/* Definição de Constantes */
#define   NOME_SISTEMA     "SCHOT - Sistema de Controle de Reservas"
#define   MAXANDARES        10    { Numero de andares do hotel }
#define   MAXQRTOSANDAR  4  /*Nr quartos em cada andar do hotel}

/* Definição de Tipos */
typedef   enum {  RESERVADO,    OCUPADO,
                  VAGO,             NAODISPONIVEL } TSTATUS;
typedef
  struct {  short        nrQuarto; /* 101..404 no exemplo */
            TSTATUS  status;
            char         nomeHosp[30];
            char         dataEntrada[9];  /* AAAAMMDD - Ex: 19990713 */
            char         dataSaida[9];
} TQUARTO;
```

Exemplo de uso:

```
TQUARTO Hotel[MAXANDARES][MAXQRTOSANDAR];
Hotel[0][0].nrQuarto = 101;
```

5. <u>Modelagem de uma Biblioteca</u>

```
/* Definição de Constantes */
#define  ARQ_BIB            “BIB.DIC”
#define  ARQ_IND_BIB     “BIB.INX”
#define  NOME_SISTEMA  “SCBIB - Sistema de Controle de Biblioteca”
#define  MAXLIVROS        500   /* Máximo de livros disponíveis na biblioteca */

/* Definição de Tipos */
typedef  short   TINDLIVRO;    typedef  char    TASSUNTO[21];
typedef  char    TISBN[12];      typedef  char    TTITULO[16];
```

```
typedef  char    TAUTOR[16];  typedef  char    TEDITORA [16];
typedef  enum{ REMOVIDO,  PRESENTE }  TSTATUS;

typedef
  struct {     TISBN          isbn;       /* Chave Primaria */
               TAUTOR       autor;
               TTITULO      titulo;
               TASSUNTO   palChave[3];
         } TIDLIVRO;              /* Informações de identificação de um livro */

typedef
  struct {     TSTATUS     status; /* Indica se registro foi ou não deletado */
               TIDLIVRO     id;
               TEDITORA    editora;
                short          nrPags;
         } TLIVRO;

typedef
  struct {     TINDLIVRO    nrReg; /* Numero de registro fisico do livro em disco */
               TIDLIVRO     id;
         } TINFOPSQ;       /* Informações úteis para pesquisa de livro no arquivo */


typedef
  struct   {  TINDLIVRO    nrTotLivros;       /* Nr Total de Livros cadastrados  */
               TINFOPSQ    index[MAXLIVROS];  /* Vetor índices dos livros */
    } TBIBLIOTECA;
```

Exemplo de uso:

```
TBIBLIOTECA     bib;
bib. nrTotLivros = 10;
strcpy(bib.index[0].id.autor, "Carl Sagan");
```

Exercício: Observe que em C não existe nenhum mecanismo de proteção dos dados do TAD, a não ser a própria disciplina do programador. Refaça estes exemplos agora utilizando C++ e Java.

4.5 Estruturas de Controle

Além da instrução de atribuição, que associa valores a determinadas posições de memória através do que chamamos de ***Binding de Nome***, toda LP apresenta estruturas de controle para dar maior flexibilidade ao programador. Elas se dividem em: condicional, iterativa e desvio incondicional.

1) **Condicional**: determina bloco de execução baseado em testes. Existem 4 formas de estrutura condicional: ***condicional simples, condicional com 2 alternativas, condicional multi-alternativa e condicional não-determinística***.

2) **Estruturas Iterativas**: ***iteração infinita, iteração com teste a priori, iteração com teste a posteriori, iteração in-Teste, iteração com número fixo de passos e iteração não-determinística.***

3) **Desvio Incondicional**.

4.5.1 Condicional Simples

Consiste de um teste único que determina se um bloco deve ou não ser executado.

```
if <expressão booleana> then <bloco de comandos>
```

Variação: delimitação do bloco de comandos:

- ALGOL 60, Pascal: **begin ... end;**
- C,C++,Java: **{ ... };**
- ADA: **endif (keyword)**;
- MODULA 2: **end (keyword)**

4.5.2 Condicional Com 2 Alternativas

```
if <expressão booleana> then <bloco de comandos>
else <bloco de comandos>
```

4.5.3 Condicional Multi-alternativas

Existem 2 formas possíveis:

Seqüência de expressões booleanas: a primeira expressão verdadeira determina o bloco de comandos a ser executado.

```
if <expressão booleana> then <bloco de comandos>
else if  <expressão booleana> then <bloco de comandos>
        .........
else <bloco de comandos>
```

Comando CASE: avalia uma expressão e executa bloco correspondente àquele valor. Obs: é mais restrito que o caso anterior ! Porque ?

```
case <expressão> of
        <val1>: <comando1>
        <val2>: <comando2>
        ...
        <valN>: <comandoN>
end
```

Em Pascal: onde <expressão> pode ser qualquer tipo discreto (int, char, tipo enumerado) e <val1> ... <valN> são listas de valores.

```
switch (<expressão seletora>) {
        case <expressão> : [<seq. de comandos>]
         [default: <seq. de comandos>]
}
```

Em C: onde: <expressão seletora> é interpretada como um inteiro; Cada expressão case só pode especificar um único valor;

4.5.4 Condicional Não-Determinística ou Comando Guardado

É uma extensão da condicional multi-alternativa proposta por Dijkstra (1975).

```
if <cond1> <seqüência de comandos 1>
when <cond2> <seqüência de comandos 2>
        .........
when <condN> <seqüência de comandos N>
fi
```

Todas as condições são avaliadas (e não até encontrar a primeira verdadeira). Quando mais de uma for satisfeita, a seqüência de comandos a ser executada é escolhida de forma não-determinística. Ou seja, não há uma regra para escolher uma delas - qualquer uma pode ser escolhida. Se nenhuma condição for satisfeita, um erro ocorre.

As condições são chamadas "guardas" ou "sentinelas". ADA tem uma versão desse comando e a usa para implementar controle de concorrência.

4.5.5 Iteração Infinita

```
do forever
        <seqüência de comandos>
end do
```

```
Simulação em Pascal:
while true do begin
   <seqüência de comandos>
end;

Em ADA: (comando LOOP)
LOOP
   <seqüência de comandos>
end LOOP;
```

4.5.6 Iteração com Teste a Priori

```
Em Pascal: while <condição de continuação> do
                <corpo_iteração>
Em C: while (<expressão>)
          <corpo_iteração>
```

4.5.7 Iteração com Teste a Posteriori

```
Em Pascal: repeat
                <corpo_iteração>
             until <condição de terminação>
Em C: do
            <corpo_iteração>
         while (<expressão>);
```

Simulação em ADA:

```
LOOP
  <corpo>
  exit when <condição>;
end LOOP;
```

4.5.8 Iteração In-Teste

Teste no meio do corpo de iteração (caso bem justificado de "goto")

```
Em ADA: (através do comando exit [when <condição>])
        LOOP
          <corpo_iteração1>
          exit when <condição>
          <corpo_iteração2>
        end LOOP;

Em C, C++, Java: (através dos comandos break e continue)
        do {
          ...
          if ( cond0 )
             continue;   /* Salta para a próxima iteração loop */
          do {
             ...
             if (cond1)
               break;       /*  termina execução do do interno */
           } while (cond2);
        } while (cond3);
```

Extensão: aninhamento

```
LOOP
  <corpo_A>
  LOOP
    <corpo_B>
    exit when <condição> (sai do loop mais interno)
    <corpo_C>
  end LOOP;
  <corpo_D>
end LOOP;
```

ou

```
OUTER: LOOP (forma de rótulos exclusiva desse comando)
       <corpo_A>
       LOOP
         <corpo_B>
       <- exit OUTER when <condição>
         <corpo_C>
       end LOOP;
       <corpo_D>
   -> end LOOP OUTER;
```

4.5.9 Iteração com Número Fixo de Passos

O controle é feito por VC (Variável de Controle) e a forma mais geral é:

for <VC> := <V. inicial> to <V. final> step <incremento>

do <corpo> ou

for (<V. in.>; <V. Contin.>; <expr. mod.>) <comandos>;

Onde todas as expressões são do mesmo tipo de VC.

Variações entre as linguagens:

- Tipos permitidos para VC:
  - ADA, Pascal: integer, char, enumerados
  - FORTRAN: integer, real
  - C: integer
- Escopo de VC:
  - ADA: equivale à declaração local
  - Pascal, C, FORTRAN: escopo da declaração

VC pode ser alterada no corpo da iteração?

Em geral, não.

Valor de VC depois do termino da iteração:

- ADA: não haverá Binding de locação e valor
- Pascal: indeterminado
- C: valor final incrementado

Quando as expressões final e de incremento são avaliadas?

Em Pascal:

```
for i := 1 to n (I) do
  n := n + 1;
```

Se **(I)** for reavaliada a cada passo, essa iteração seria infinita. Em geral, as 2 expressões (final e incremental) são avaliadas uma única vez, no início da iteração. Assim, a modificação de n, acima, não alteraria o número de passos inicial.

Formas de incremento:

- ADA, Pascal: "incremento de 1" (succ)
- C, FORTRAN: expressões

Decrementos:

- Pascal: **for i := 6 downto 1 do ...**
- ADA: **for i in reverse 1..6 loop ... end loop**

Desvios para o interior do loop

- ADA, FORTRAN: proíbem
- Pascal: permite

4.5.10 Iteração Não-Determinística

```
do
  when <cond1> <seqüência de comandos1>
  when <cond2> <seqüência de comandos2>
   ...
  when <condN> <seqüência de comandosN>
od
```

Repete enquanto pelo menos uma condição for verdadeira. Se mais de uma é verdadeira, a escolha da seqüência é feita não-deterministicamente.

4.5.11 Desvio Incondicional

```
goto <label>
```

Módula 2 não tem esse comando.

Labels:

- Pascal: declarados
- C, FORTRAN, ADA: implícitos
- PL/I, SNOBOL, APL: variáveis (podem ser passados como parâmetros e permitem arrays da labels)

4.6 Tipos de Dados

4.6.1 Checagem de tipo

A checagem de tipo consiste na determinação do tipo de um certo dado objeto. Pode ocorrer em tempo de compilação, de execução ou pode não ocorrer. LPs com checagem de tipo em tempo de compilação são ditas fortemente tipadas (por exemplo, ADA). Pascal, por exemplo, não é fortemente tipada: a maior parte da checagem é feita em tempo de compilação, mas os subintervalos e os registros variantes não são checados. Quando não há possibilidade de haver checagem em tempo de compilação em determinadas circunstâncias, há duas alternativa: ou não se faz a checagem (Pascal), ou se faz em tempo de execução - o que pode ser muito caro em tempo (cada vez que o dado objeto é referido) e em espaço (um indicador de tipo deve ser armazenado como parte de cada valor de dado).

LPs dinamicamente tipadas, onde os Bindings são feitos em tempo de execução, fazem checagem apenas em tempo de execução, enquanto que LPs estaticamente tipadas, com Bindings em tempo de compilação, podem checar tipos ou em tempo de compilação ou em tempo de execução.

Exemplo da checagem em Pascal:

```
Type soft = record
            case test: boolean of
true: (first: 1..20);
false: (second: char);
  end;
var x, y: soft;
   c: char;
begin
...
c := x.second;
y.first := 2 * x.first;
...
end.
```

Pascal não pode verificar totalmente os tipos por 2 razões:

1. Quando x.second é usado, não é possível checar, durante a compilação, se a segunda parte variante de x está ativada. (se x.teste = true, então x.second tem "lixo")

2. Na atribuição a y.first, haverá violação se x.first 10, o que não é possível verificar em tempo de compilação.

4.6.2 Conversão de Tipos

A conversão de tipos pode ser implícita (coerção de tipos) ou explicita. Por exemplo, há conversão implícita entre um "integer" e um "real" e explicita quando se usa float(), round, trunc.

Quando dois tipos são considerados iguais?

Sejam os tipos e variáveis:

```
type   T1: 0..10;
       T2: 0..10;
var    A, B, C: T1;
       D: T2;
```

São legais as seguintes atribuições?

```
A := B;
B := C;
A := D;
```

Existem 3 perspectivas para se estabelecer a igualdade entre tipos:

1. **Equivalência de domínios**: dois dados objetos são equivalentes se tiverem associados a seus tipos um mesmo domínio de valores possíveis (equivalência estrutural). Neste caso A := B; A := D; e B := C são legais (A, B, C, D são tipos equivalentes). É o que ocorre em ADA.

2. **Equivalência de nome**: dois dados objetos são de tipos equivalentes se seus tipos tiverem o mesmo nome. Neste caso A, B e C são variáveis de tipos equivalentes, mas D não é. É o que ocorre em Pascal.

3. **Equivalência de declaração**: dois dados objetos são de tipos equivalentes se eles são ligados ao tipo numa mesma declaração. Neste caso só B e C são de tipos equivalentes.

E os tipos anônimos?

Por exemplo (ADA, Pascal):

```
var E, F: 0..10;

    G: 0..10;
```

Neste caso não existe equivalência de nome.

Subtipos e Tipos Derivados

Por exemplo:

```
var A: integer;  B: 0..100;

     A := B é válido? e B := A
```

O subtipo inclui o conjunto de operadores do tipo principal. Nestes casos a checagem é realizada em tempo de execução.

Forçando incompatibilidade em LP com equivalência de domínio

(ADA): type tempo is new float; comprimento is new float;

duração: tempo; distância: comprimento;

Neste caso o domínio de "duração" é igual ao domínio de "distância" mas a operação "duração + distância" não é permitida.

4.6.3 Tipos de Dados Escalares

São aqueles dos dados objetos atômicos, que não podem ser desmembrados em componentes. Podem ser Built-in types (já presentes na LP) ou User-Defined types (definidos pelo usuário).

Propriedades a considerar:

1. Parâmetros do tipo (por exemplo, precisão)
2. Formato da declaração
3. Domínio
4. Operações (operadores e funções)
5. Atributos
6. Conversão
7. Constantes pré-definidas
8. Implementação

As LPs em geral apresentam os seguintes tipos básicos: Tipo Numérico – Inteiro, Tipo Numérico – Real, Tipo Boolean, Tipo Ponteiro, Tipos Enumerados e Tipos Compostos.

4.6.3.1 Tipo Numérico - Inteiro

Algumas LPs suportam um único tipo inteiro (integer) cujo domínio é o intervalo dos inteiros que podem ser representados na máquina. A desvantagem desta implementação é que tem o domínio dependente de implementação.

Outras LPs permitem a especificação de subtipos de inteiros, limitando o domínio a um intervalo do domínio de inteiros da máquina.

Exemplos:

- Módula 2: "integer" e "cardinal" (não-negativos)
- C: "short int", "int", "long int", "unsigned"
- Pascal: type Positivos = 1..maxint
- ADA: subtype T is Integer range 0..maxint (o tipo é derivado do tipo das constantes do intervalo)

A checagem de tipos para os subtipos pode ser de duas maneiras:

- ignorar a checagem, já que há necessidade de checagem em tempo de execução - o que custa tempo.
- checar, em tempo de execução, toda vez que um novo binding de valor é feito

A escolha entre essas opções pode ser feita em tempo de compilação.

Implementação

Geralmente a linguagem segue a representação de inteiros da máquina em que a LP é implementada. (em geral binário com complemento de 2)

Operações

São em geral implementadas diretamente em linguagem de máquina (LM) ou podem ser programadas em LM usando algoritmos aritméticos padrões.

4.6.3.2 Tipo Numérico - Real

Possíveis parâmetros:

- número de dígitos significativos
- fator de escala designando o número de dígitos à direita do ponto decimal

Exemplo: Números reais com 10 dígitos significativos e fator de escala de 2 dígitos representariam todos os números de -99999999.99 até 99999999.99 com intervalo de 0.01 de distância (representação em ponto fixo).

A maioria das LPs utiliza representação com ponto flutuante (mantissa e expoente). ADA utiliza tanto ponto fixo como ponto flutuante.

4.6.3.3 Tipo Boolean

Domínio = {Falso, Verdadeiro}

Em C representado como 0 e 1 (boolean é um subtipo de integer)

Operadores: AND, OR, NOT

Considere: (I <> 0) AND (K/I > 6)

Se I <> 0 for falso, então K/I não pode ser executado, desde que I é zero.

Mas, se a primeira condição é falsa, então não há necessidade de avaliar a segunda, já que a conjunção será falsa. Então uma implementação mais esperta prevê:

A AND B = if A then B else FALSE

e

A OR B = if A then TRUE else B

ADA possui AND THEN e OR ELSE - opcional para programador - além de AND e OR.

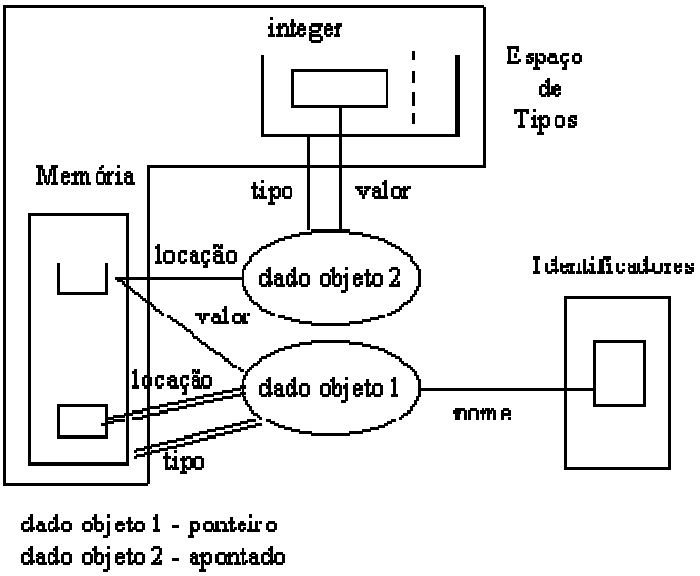
Constantes FALSE e TRUE.

4.6.4 Tipo Ponteiro

O valor de um ponteiro é o endereço de um outro dado objeto. Neste caso existem dois dados objetos envolvidos: o apontador e o apontado.

Neste caso, os bindings de tipo e nome ocorrem em tempo de compilação, o binding de locação ocorre em tempo de loading e o de valor, em tempo de execução.

Bindings para ponteiros



O binding estático de tipo requer a declaração do tipo do objeto apontado (ADA, PASCAL, C), já no binding dinâmico de tipo o tipo do objeto apontado é derivado do valor a ele atribuído em tempo de execução.

Exemplo:

- PASCAL: var P: ^ tipo;
- ADA: P: access ^ tipo;
- C: int * P;

O binding de valor a um ponteiro é feito de duas formas:

1. criação
   - Pascal - new (P);
   - ADA - P := new tipo; ou P := new integer'(0); - cria e aponta para o inteiro 0
   - C - malloc (P);
2. atribuição
   - Pascal - P := Q; - onde var P, Q: ^tipo;
   - C (operadores de ponteiros) - int Y; int *P = &Y; int X; *P = X;
3. Desalocação de objetos apontados em tempo de execução:
   - Pascal - dispose (P);
   - ADA, C - free (P);

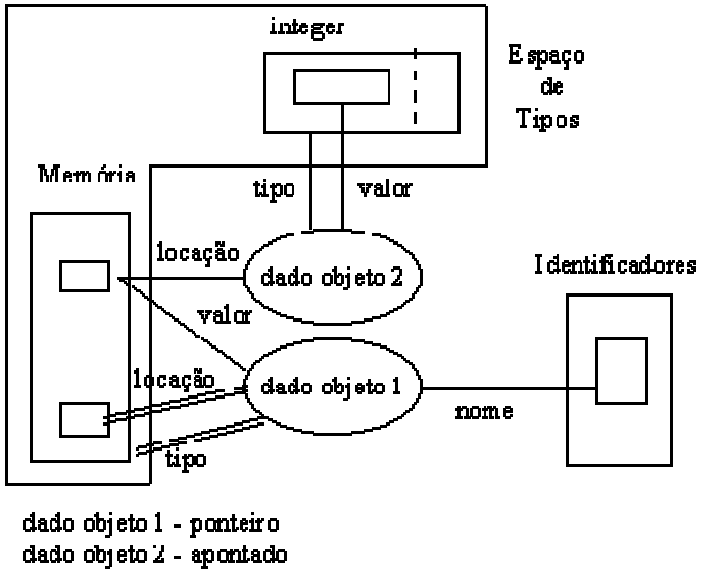
Atenção (desalocação implícita x explícita) : uma vez desalocado o objeto apontado, é desfeito o binding de valor do objeto ponteiro - o que inviabiliza o uso do elemento apontado. Esses ponteiros são ditos estar "suspensos". (Mas eles podem ter restabelecido o binding de valor através de uma atribuição). O objeto apontado desalocado é dito garbage, impossível de ser alcançado pelo programa. (**Garbage Collection - Alocação dinâmica**)

4. Deferência em ponteiros

"Deferenciar" um ponteiro é o processo de obter o valor do objeto apontado, através da referência do nome do ponteiro.

Exemplo:

- Pascal - P^
- ADA - P.ALL
- C - *P



5. Constantes utilizadas para ponteiro nulo:
   - Pascal - NIL
   - ADA, C – NULL

4.6.5 Tipos Enumerados (domínio criado pelo usuário)

O usuário cria e dá nome a uma lista finita de valores. Em Pascal por exemplo:

type cor = (verde, azul, branco);

var c: cor;

Operadores: atribuição e comparação (ordem em que aparecem na lista - verde < azul < branco). Em Pascal existe também succ( ) e pred( ).

Implementação: mapeamento nos inteiros não negativos 0, 1, 2, ...

Dificuldades:

1) Mesmo identificador em tipos distintos

Exemplo:

type fruta = (maçã, laranja, pêra); cor = (branco, azul, laranja);

var F: fruta; C: cor;

Pascal proíbe, ADA permite e trata como:

Exemplo:

f := laranja - fruta (pelo contexto - F é do tipo fruta)

Em contextos ambíguos, permite o uso da forma F := fruta'laranja.

4.6.6 Tipos de Dados Compostos

Existem várias formas de composição, correspondendo aos diversos tipos de dados compostos:

- **Mapeamento**: f: T1 -> T2 (arrays)
- **Produto Cartesiano**: T = T1 x T2 x ... x Tn (records)
- **União discriminante**: T = T1 U T2 U ... U Tn (records variantes)
- **Seqüência**: T = {(t1, t2, ..., tn)} (strings, files)
- **Conjunto Potência**: T = 2T1 (sets)

4.6.6.1 Arrays

<def. array> ::= array [<tipo domínio>] of <tipo base>

onde <tipo domínio> é finito e <tipo base> é qualquer.

Em Pascal, FORTRAN e C o binding de <tipo domínio> é feito em tempo de compilação. Isso determina a impossibilidade de usar parâmetros, do tipo array, com domínios gerais.

```
Exemplo: type   TipoA1 = array [1..10] of integer;
                TipoA2 = array [1..20] of integer;
         Var    A1: TipoA1;
                A2: TipoA2;
        procedure media1(A: TipoA1, var media: integer);
        procedure media2(A: TipoA2, var media: integer);
        ...
        media1(A1, m);
        media2(A2, m);
```

Em ADA a definição é irrestrita.

Exemplo:

```
type vector is array (integer range <>) of float;
var V: vector (INF..SUP); {declaração local a um bloco}
```

Seleção:

a[i] , a(i) , slice: a (3..N) - ADA

Construção:

ADA: (1.0, 5.0, 3.0, 6.0)

Extensão: (1 -> 7.0, 5..12 -> 1.0, others -> 0.0)

C: int name[10] = {6, 4, 8, 9, 2, 7, 17, 21,16, 8}

Atribuição:

ADA: V := (1.0, 3.0, 5.0)

V1 := V2 (se tipos_bases e cardinalidade dos tipos_domínio forem iguais)

V1 is array (1..10) of integer;

V2 is array (11..20) of integer;

Operadores de agregação:

APL: + (14, 12, 1) = 27

Atributos derivados da estrutura:

C: sizeof()

ADA: first, last, range, length

Exemplo: type TipoA is array (integer range <>) of character;

A: TipoA (4..12);

A'first é 4

A'last é 12

A'range é 4..12

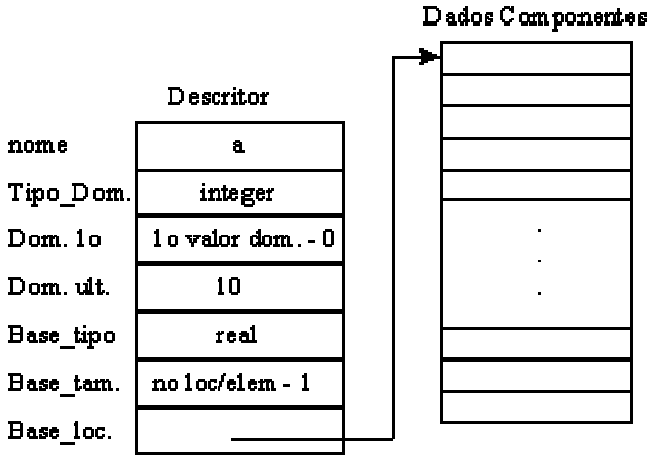
A'length é 9

Comparação: (ordem lexicográfica)

ADA, Pascal.

Implementação:

Exemplo:



Endereço I = Base_loc + (I - Dom_1o) * Base_tam

Extensão para arrays multidimensionais - armazenamento por linha ou coluna.

4.6.6.2 Records ou Structures

Pascal:

```
type T = record
           C1: integer;
           C2: real;
         end;
var r: T;
```

C:

```
struct T { int C1;
           float C2;   }
struct T r = {10, 5.7}
```

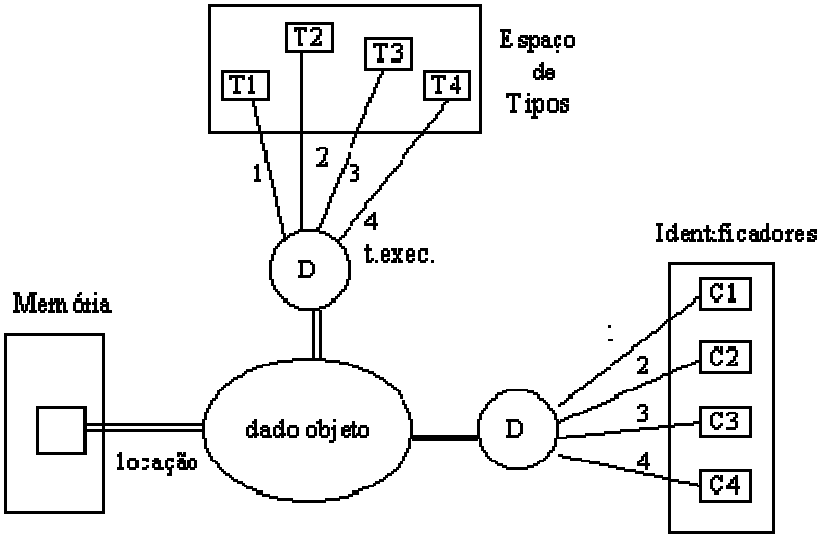
Obs: Pode haver comparação de registro de igual estrutura (igualdade e desigualdade).

4.6.6.3 Records Variantes

Modelo: união discriminante (presença de um componente que determina qual dos conjuntos de tipos é ativo num dado momento).

ADA:

```
type T is record
        case D: 1..4 is
          when 1 -> C1: T1;
          when 2 -> C2: T2;
          when 3 -> C3: T3;
          when 4 -> C4: T4;
        end case;
      end record;
```

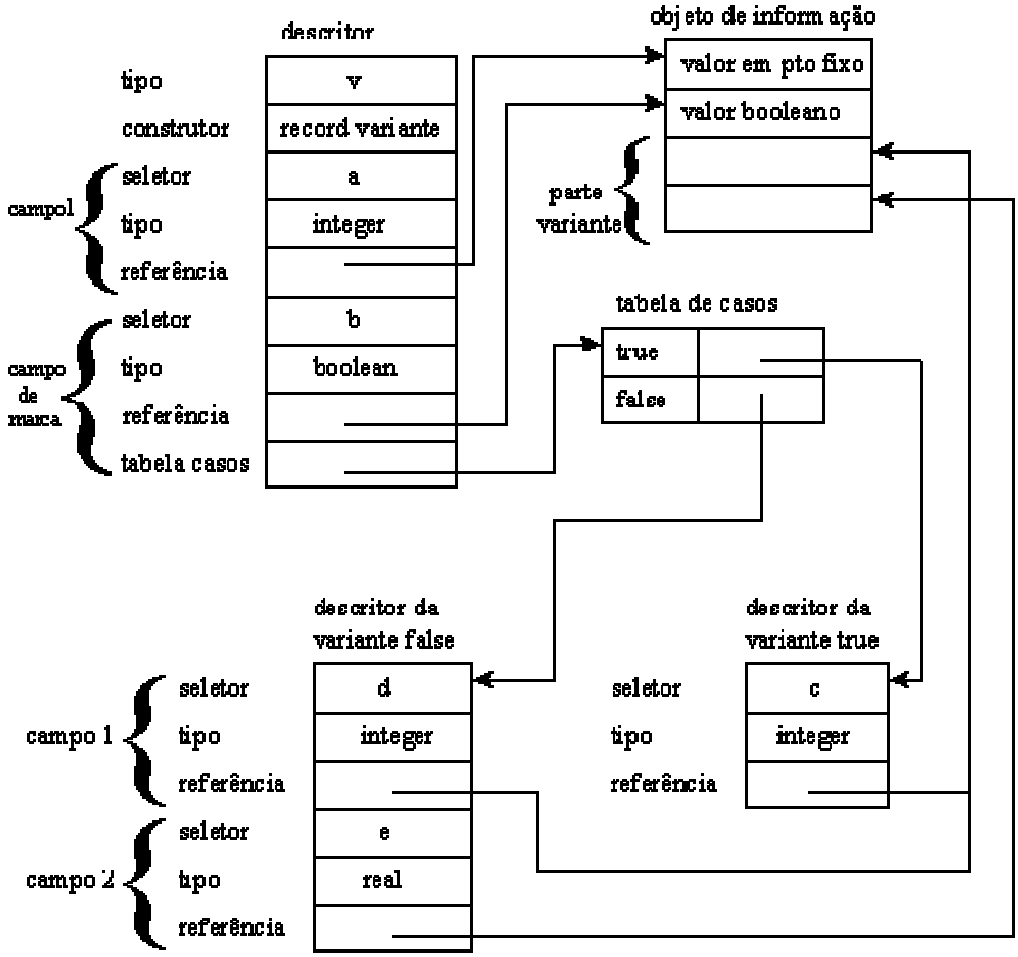


Pascal:

```
type V = record
           a: integer;
           case b: boolean of
             true: (C: integer);
             false: (D: integer;
                     E: real);
         end;
```

Em C : unions - indiscriminadas

Representação:

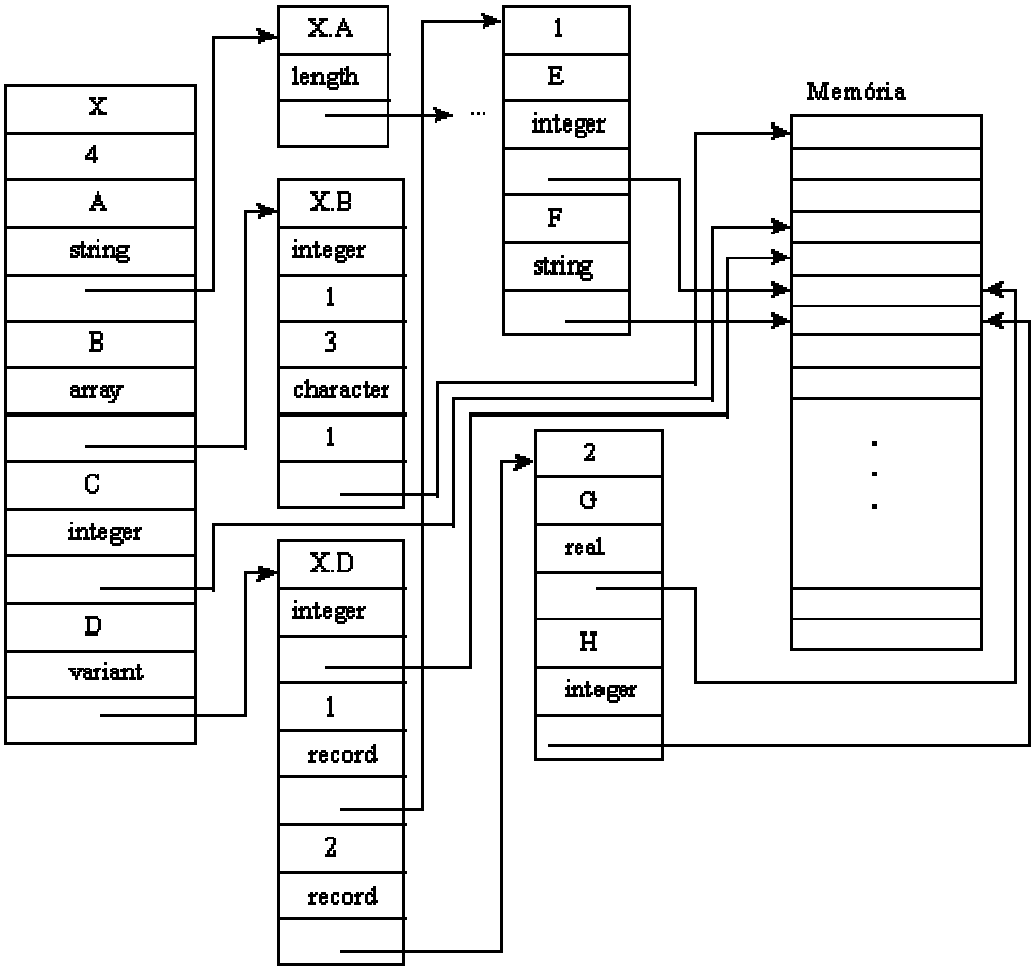


Implementação: A memória reservada para parte variante corresponde ao maior espaço necessário para armazenar qualquer alternativa.

Agregados Aninhados

Ex. em ADA:

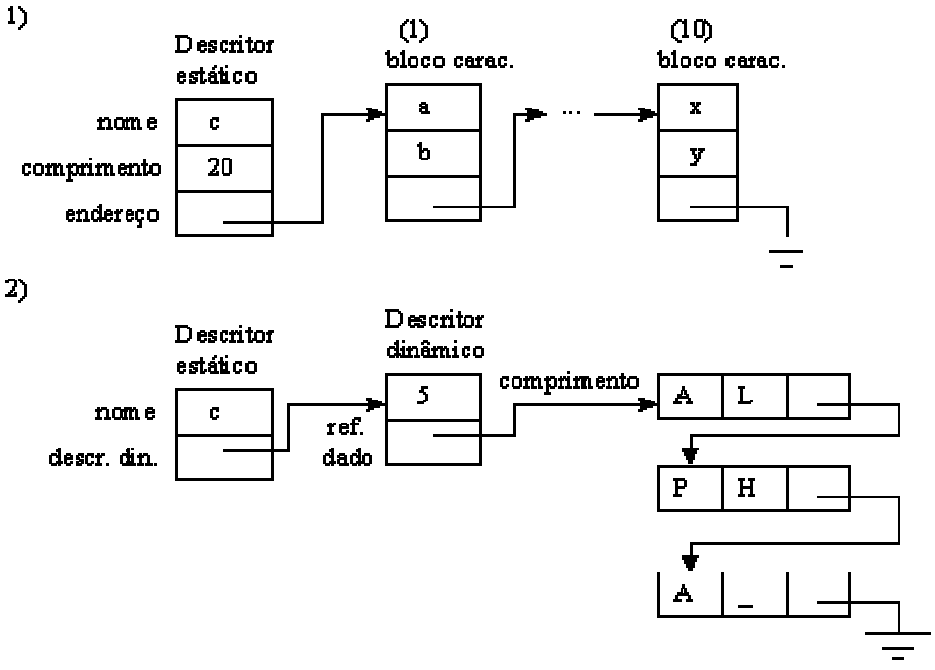
```
X: record
   A: string;
   B: array (1..3) of character;
   C: integer;
   case D 1..2 is
      when 1 -> E: integer;
                F: string;
      when 2 -> G: real;
                H: integer;
   end case;
end record;
```



4.6.7 Strings

Existem 2 tipos de modelagem:

1) Como arrays de caracteres (Pascal, C, ADA), possuindo tamanho (máximo) fixo, e inexistência de operadores próprios (funções).



2) Como seqüências "dinâmicas" (SNOBOL, BASIC), possuindo tamanho ilimitado e operadores próprios (concatenação, substring). Neste caso o binding é feito em tempo de execução.

4.6.8 Arquivos (Files)

Arquivos ou Files são seqüências de componentes de um mesmo tipo, armazenadas em memória secundária.

Tipo x Facilidade de E/S : Pascal: **file of <tipo_componente>**

S.O. : binding de locação (assign)

Operações: reset, rewrite, get, put, seek

seleção: <nome_arq>^, eof ().

4.6.9 Conjuntos (Sets)

Pascal: **set of <tipo_base>**

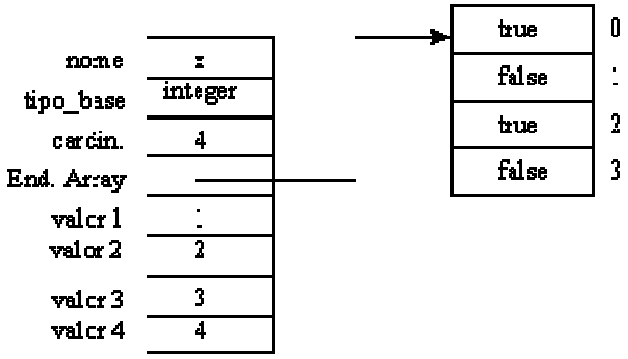
type S = set of 1..4;

Valores possíveis: [], [1], [2], [3], [4], [1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4], [1, 2, 3], [[1, 2, 4], [1, 3, 4], [2, 3, 4], [1, 2, 3, 4].

| tipo_base | = C -> | tipo_conjunto | = 2^C

Implementação: array de boolean

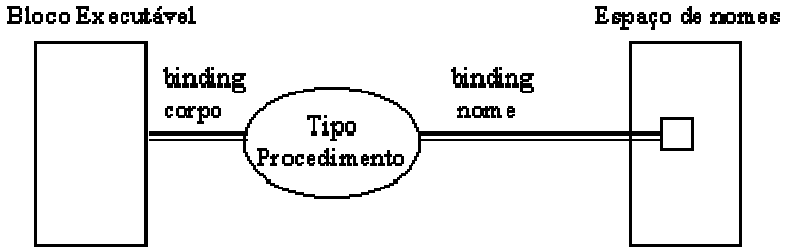
Ex: var X: S ; X := [1, 3]



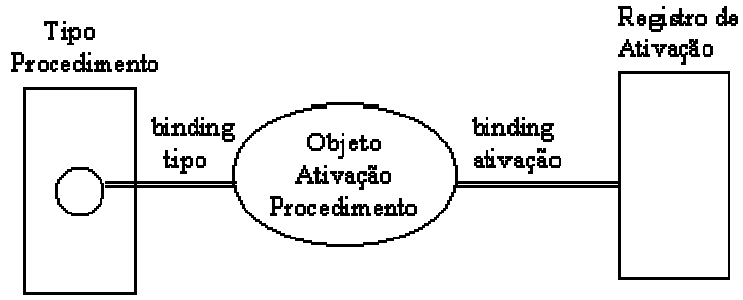
4.7 Procedimentos e Funções

Abstrações de programas -> expressões (funções) ou comandos

Analogia: Tipo de Dado <-> Tipo de Procedimento. A definição é feita em tempo de compilação.



Invocação (chamada): Novo binding do objeto (procedimento) a uma ativação do procedimento (em tempo de execução).



Analogia: Def. Tipos de Dados x Def. de Procedimentos

Assim como a cada nova Definição de Variáveis (a cada entrada do bloco) é criado um objeto, também na Ativação de um Procedimento (a cada chamada) é criado um objeto de ativação, chamado **registro de ativação (R.A)**

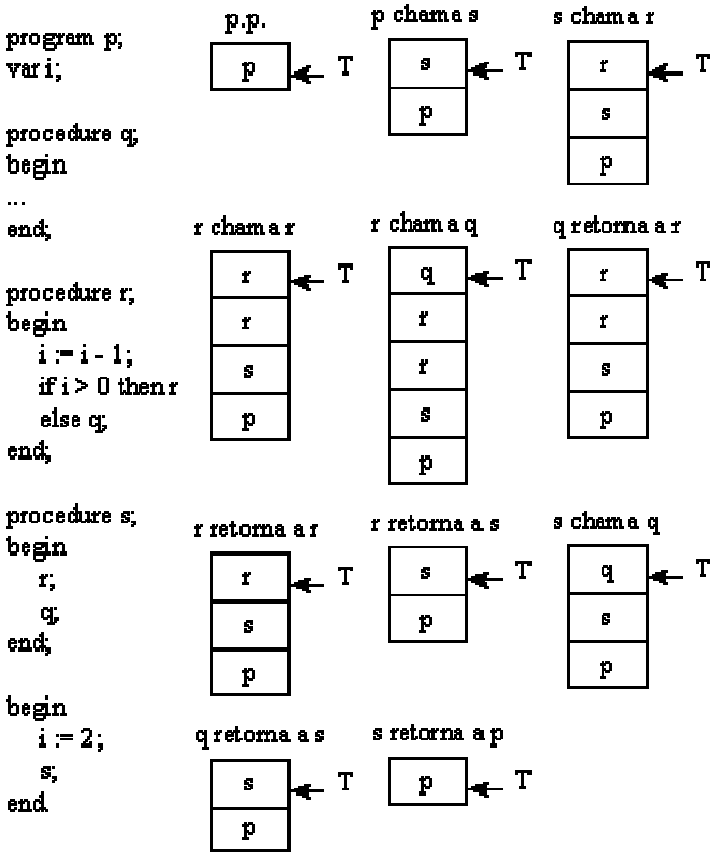
A forma geral de um R.A. consiste de três partes:

- **ambiente local** (objetos definidos localmente no procedimento)
- **ponteiro para ambiente global** (ponteiro para registro de ativação do qual a ativação atual herda objetos e seus bindings - objetos globais)
- **ambiente de parâmetros** (informação sobre os objetos passados para e a partir do procedimento)

4.7.1 Pilha de R.A.

Quando um procedimento é chamado, seu R.A. é empilhado numa pilha de R.A., durante o tempo de execução. Quando ele termina, seu R.A. é desempilhado, fazendo com que a unidade que o chamou passe a ter seu R.A. no topo da pilha e portanto, passa a ser a unidade ativa atual. Assim, a pilha de R.A. representa todas as unidades de programa que estão em "processo de execução" (apenas a do topo está sendo executada no momento).

Exemplo:



Chamadas recursivas geram R.A. distintos **ambientes locais**.

4.7.2 O ambiente local do RA

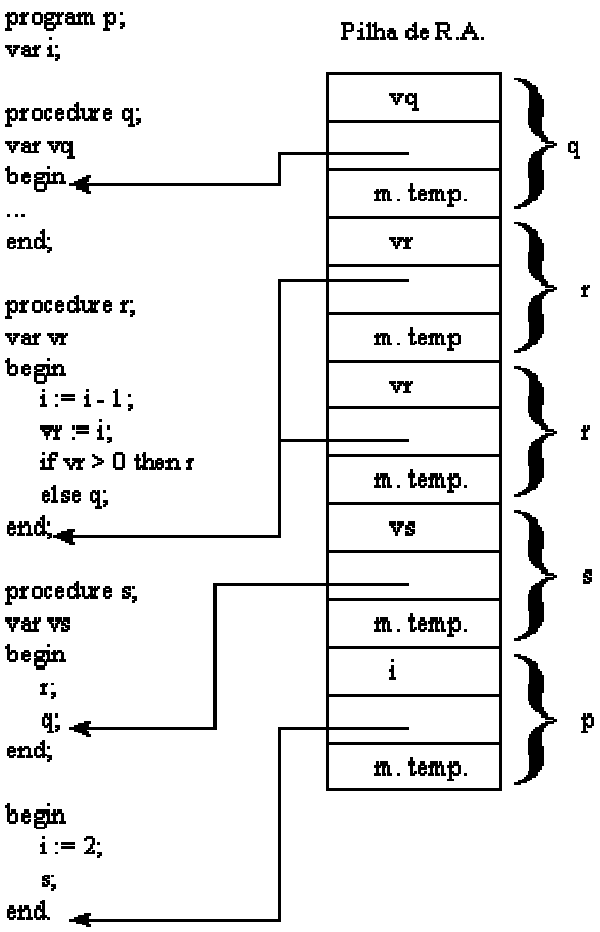
O ambiente local inclui:

a) os objetos declarados localmente no procedimento

b) um ponteiro para a próxima instrução a ser executada no procedimento (endereço de retorno) - cada ativação possui um registro de ativação, assim, há múltiplos endereços de retorno

c) memória temporária para reter dados durante avaliação de expressões. (ex: chamada de função durante avaliação de expressão) -> dependente de implementação.

Estendendo o exemplo anterior com o ambiente local (instante p -> s -> r -> r -> q da pilha):



4.7.3 O ambiente global do RA

Permite acesso a objetos definidos fora do procedimento. O ambiente onde esses objetos são encontrados é representado no R.A. por um ponteiro para o R.A. cujos ambientes local e global irão determinar o ambiente global do R.A. atual. Ou seja, quando um objeto não é encontrado localmente (na porção do ambiente local do R.A.), uma busca é efetuada a partir do ponteiro do ambiente global deste R.A..

Possíveis definições de ambiente global:

- Escopo Estático
- Escopo Dinâmico

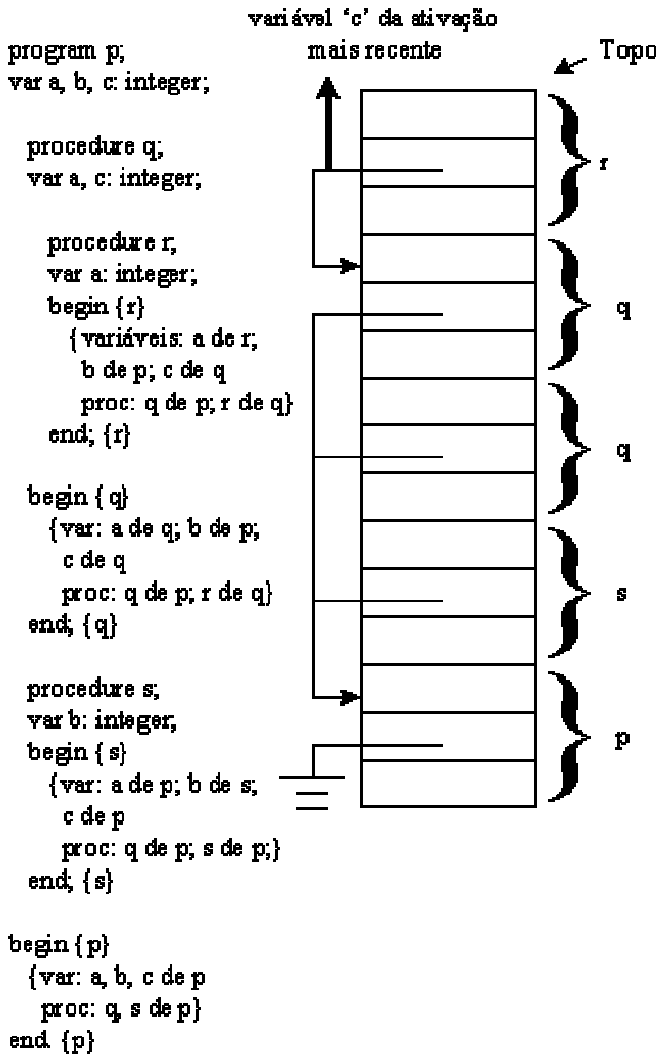
Uma terceira alternativa é a utilização unicamente de ambiente local. Com isso elimina-se riscos de efeitos colaterais (alteração de um ambiente sem mudança especificada no código). No FORTRAN todas as variáveis são locais, exceto as definidas com o comando COMMON. Em C todas as variáveis declaradas fora de procedimento são globais. O C apresenta outras categorias de variáveis: automatic, external, static.

4.7.3.1 Escopo Estático

O ambiente global é herdado da unidade em que o procedimento foi definido. Portanto um nome usado no procedimento, mas não "ligado" (definido) nele, herda o binding que este nome tinha na unidade de programa que contém (imediatamente) o procedimento. (= definição de escopo estático de blocos).

A *definição do ambiente global é feita em tempo de compilação*, já que depende unicamente da posição da definição do procedimento. No entanto, ele é especificado em tempo de execução no R.A. porque ele precisa se referir a um R.A. específico da unidade que o contém.

Exemplo:



4.7.3.2 Escopo Dinâmico

O *ambiente global é determinado pelos ambientes local e global da unidade que chamou o procedimento.* Sua determinação é feita em tempo de execução.

Vantagem: eliminação do ponteiro para ambiente global (implícito na pilha).

Desvantagem: ilegibilidade e fonte de erros.

Exemplo:

```
program p;
var a: integer;

  procedure q;
  begin {q}
    {var a de p out}
  end; {q}

  procedure r;
  var a: integer;
  begin {r}
    q;
  end; {r}

begin {p}
  q;
end. {p}
```

4.7.4 O ambiente de parâmetros

Parâmetros são o meio de comunicação entre unidades de programa. Existem 3 categorias de parâmetros:

- **Parâmetro de Entrada (IN):** informação da unidade que chamou para o procedimento.
- **Parâmetro de Saída (OUT):** informação do procedimento para a unidade que o chamou.
- **Parâmetro de Entrada/Saída (IN/OUT):** informação em ambos os sentidos.

Parâmetro atual é aquele que aparece na chamada do procedimento (valor - IN, endereço - OUT ou IN/OUT) e **parâmetro formal** é aquele que aparece na definição. A associação dos parâmetros pode ocorrer em tempo de compilação ou em tempo de execução.

Em C todos os parâmetros são IN e implementados por cópia (referência pode ser feita via ponteiros). Em Pascal: IN/OUT - referência (var), IN - cópia (default)

Métodos de Associação de Parâmetros:

- **posicional**: posições nas listas
- **por nome**: nome do parâmetro formal é associado ao atual na chamada

Exemplo:

ADA utiliza os dois métodos e ainda um misto:

procedure TEST (A: in TipoA; B: in out TipoB; C: out TipoC);

Associação posicional: TEST (x, y, z);

Associação por nome: TEST (A->x, C->z, B->y);

Associação mista: TEST (x, C->z, B->y);

Existe ainda a associação por default: (para parâmetro IN) na ausência de parâmetro atual, assume-se valor default especificado para parâmetros formais.

4.7.4.1 Parâmetros de Entrada (IN)

Podem ser implementados de duas maneiras:

- **Referência:** o endereço do parâmetro atual torna-se o endereço do parâmetro formal. Modificações no parâmetro formal alteram o atual. Neste caso deve-se haver uma verificação de não alteração: não pode aparecer do lado esquerdo de atribuição e como parâmetro OUT ou IN/OUT em outro procedimento. Neste caso o parâmetro deve ser tratado como constante no procedimento.

- **Cópia (valor):** o parâmetro formal é tratado como variável local inicializada com o valor do parâmetro atual. Modificações no parâmetro formal não afetam o parâmetro atual. Este método é menos eficiente em tempo (cópia) e espaço (variáveis locais), porém é mais flexível, permitindo a modificação dos parâmetros formais e um menor número de variáveis locais.

4.7.4.2 Parâmetros de Entrada e Saída (IN/OUT)

Podem ser implementados de duas maneiras:

- **Referência:** não há restrições ao uso do parâmetro formal no procedimento. Os parâmetros formal e atual compartilham o mesmo endereço.

- **Cópia (valor):** uma variável local é criada para o parâmetro formal e inicializada com o valor do parâmetro atual. Ao final do procedimento o valor final dessa variável é copiado no endereço do parâmetro atual.

Nos dois casos o efeito final é o mesmo, com locações diferentes.

Problema: "Aliasing" - referência a uma mesma locação por diferentes nomes.

Exemplo em Pascal:

```
Program main;
var A: integer;
   procedure test(var X, Y: integer);   {var -> IN/OUT}
      begin
         X := A + Y;
         writeln(A, X, Y);
      end;
begin
  A := 1;
  test(A, A);
  writeln(A);
end.
```

Se a implementação dos parâmetros for por referência o resultado deste programa será

```
2 2 2
2
```

porém ser a implementação for por valor, o resultado será

```
1 2 1
? - depende da ordem da cópia de X, Y em A
```

4.7.4.3 Parâmetros de Saída (OUT)

São pouco usados. Podem ser implementados de duas maneiras:

- **Referência:** compartilham endereço. Deve haver verificação para que o valor do parâmetro atual não seja usado no procedimento; apenas modificações ao parâmetro formal são permitidas.
- **Cópia (valor):** parâmetro formal é uma variável local com valor inicial indefinido. No final do procedimento, o valor do parâmetro formal é passado ao parâmetro atual.

4.7.4.4 Procedimentos como parâmetros

Ao especificar o parâmetro (procedimento) formal, é necessário que os tipos de todos os seus parâmetros também sejam especificados a fim de que a checagem de tipos possa ser feita quando o procedimento (formal) for chamado.

Exemplo em Pascal:

```
Program main;
   var A: real;
procedure TESTPOS(X: real; procedure ERROR(MSG: string));
   begin
      if x <= 0 then error('X negativo em TESTPOS');
   end
   procedure E1(M: string);
   begin
      writeln('E1 Erro:', M);
   end;
   procedure E2(M: string);
   begin
      writeln('E2 Erro:', M);
   end;
begin
   readln(A);
   TESTPOS(A, E1);   TESTPOS(A, E2);
end.
```

Informações necessárias para o procedimento parâmetro sobre o procedimento que o tem como parâmetro:

- endereço para o código executável
- ponteiro para o R.A do ambiente global do procedimento que o contém

Estas informações são necessárias para a criação do R.A. do procedimento parâmetro.

Em C funções podem ser parâmetros de outras funções através de "ponteiros para funções". Exemplo em C: Função que soma N termos resultantes da aplicação de uma função arbitrária aos primeiros N inteiros.

```
int sum( int N, int (*F)(int) ) {
    int i,s=0;
    for (i=1; i<=N; i++)
        s = s + (*F)(i);
    return (s);
}

int square( int i )  {
    return i*i;
}
```

```
void main( void ) /*soma dos quadrados dos primeiros 10 int */
    printf("%d\n",sum(10, square));
}
```

4.7.4.5 Passagem de Parâmetro "Por Nome"

Mecanismo usado em ALGOL 60. Na chamada do procedimento ocorre um binding do nome do parâmetro atual com o parâmetro formal. Ou seja, em run-time, há uma "substituição textual" do nome do parâmetro atual pelo parâmetro formal dentro do procedimento. A locação de memória do parâmetro atual permanece a mesma durante a execução do procedimento.

Exemplo:

```
procedure swap(a, b: integer);
var temp: integer;
begin
   temp := a;
   a := b;
   b := temp;
end;
program main;
var i, j: integer
   m: array[1 .. 100] of integer;
begin
   ...
   swap(i, j);
   ...
end.
```

Se a passagem for por nome:

```
temp := j;
j := i;
i := temp;
```

Efeito: igual a parâmetro IN/OUT por referência.

Problemas:

```
swap(i, m[i]);
```

Após a substituição textual:

```
temp := i;
i := m[i];
m[i] := temp; (neste caso o "i" está modificado)
```

Compare com passagem por referência

Quando o nome do parâmetro atual coincide com o de uma variável local:

```
Program main;
var i, temp: integer;
begin
   swap(i, temp);
end.
```

Então teremos: temp := i;

```
i := temp;
temp := temp;
```

Dois objetos estão ligados ao nome temp no procedimento (um local ao procedimento, outro local ao bloco que o chamou).

Método para diferenciar do ALGOL 60: nomes de parâmetros ligados a objetos da unidade que chamou.

No exemplo:

$temp_{swap}$:= i;
i := $temp_{main}$;
$temp_{main}$:= $temp_{swap}$;

4.7.5 Funções ou Value-Returning Procedures (VRPs)

Funções são abstrações de expressões e retornam um único valor - do tipo da função Existem dois métodos para implementar o valor de retorno:

1) **Pascal -** cria-se uma pseudo-variável no ambiente local, ligada ao nome da função. Diferenças com variável local: não é declarada e só pode ser modificada, mas não acessada, dentro da função. O valor dessa variável, por ocasião do término da execução, é o valor de retorno da função.

   Custo: objeto para armazenar o valor de retorno.

2) **ADA, C -** uso do comando RETURN seguido de uma expressão. Quando este comando é executado, a expressão é avaliada, a função terminada, e o valor da expressão é o valor de retorno da função.

Os **Modos de passagem de parâmetros** são idênticos aos de procedimento, porém os "efeitos colaterais" derivados do uso de parâmetros OUT e IN/OUT, numa expressão, não são desejáveis.

4.7.6 Overloading de Procedimentos

O overloading de procedimentos permite que dois ou mais procedimentos tenham o mesmo nome se eles puderem ser distinguidos pelo número ou tipo de seus parâmetros, ou pelo tipo do valor de retorno no caso de funções.

Vantagem: procedimentos que fazem a mesma operação sobre diferentes tipos de parâmetros podem possuir o mesmo nome.

Exemplo em ADA:

```
procedure main is
R: float := 0.0;
I: integer := 0;

   function F(X: float) return integer is
   begin
      return 1;
   end;

   function F(X: integer) return integer is
   begin
      return 2;
   end;

   function F(X: float; Y: integer) return integer is
   begin
```

```
      return 3;
   end;


   function F(X: integer; Y: float) return integer is
   begin
      return 4;
   end;

   function F(X: integer) return float is
   begin
      return 5.0;
   end;

begin -- main
   put(F(R)); -- imprime 1
   put(F(I)); -- imprime 2
   put(F(R, I)); -- imprime 3
   put(F(I, R)); -- imprime 4
   R := F(I); --
   put(F(integer(R))); -- imprime 5.0
end main;
```

5 O Paradigma Orientado a Objeto

O modelo **Orientado a Objeto** focaliza mais o problema. Um programa OO é equivalente a objetos que trocam mensagens entre si. Os objetos do programa equivalem aos objetos da vida real (problema). A abordagem OO é importante para resolver muitos tipos de problemas através de simulação.

A primeira linguagem OO foi Simula, desenvolvida em 1966 e depois refinada em Smalltalk. Existem algumas linguagens híbridas: Modelo Imperativo mais características de Orientação a Objetos (OO). Ex: C++.

No modelo OO a entidade fundamental é o objeto. Objetos trocam mensagens entre si e os problemas são resolvidos por objetos enviando mensagens uns para os outros.

Linguagens Orientadas a Objeto: SMALLTALK, C++, Java

5.1 Componentes Básicos do Modelo OO

- **Objetos:** Um objeto é um conjunto encapsulado de operações e um estado que registra e lembra o efeito das operações. Um objeto executa uma operação em resposta ao recebimento de mensagem que está associada à operação. O resultado da operação depende do conteúdo da mensagem recebida e do estado do objeto quando ele recebe a mensagem. O objeto pode, como parte dessa operação, enviar mensagem para outros objetos e para si mesmo.

- **Mensagens:** São requisições enviadas de um objeto a outro, para que este produza algum resultado desejado. A natureza das operações é determinada pelo objeto receptor. Mensagens podem ser acompanhadas de parâmetros, que são aceitos pelo objeto receptor e que podem ter algum efeito nas operações realizadas. Esses parâmetros são, eles próprios, objetos.

- **Métodos:** São descrições de operações que um objeto realiza quando recebe uma mensagem. Uma mesma mensagem poderia resultar em métodos diferentes, quando enviada para diferentes objetos. O método associado com a mensagem é pré-definido. Um método é similar a um procedimento; mas há diferenças.

- **Classes:** Uma classe é um template para objetos. Consiste de métodos e descrições de estado que todos os objetos da classe irão possuir. Uma classe é similar a um TAD, no sentido de que ela define uma estrutura interna e um conjunto de operações que todos os objetos, que são instâncias da classe, possuirão. Uma classe é também um objeto, e portanto aceita mensagens e possui métodos e um estado interno. Uma mensagem que muitas classes aceitam é uma mensagem de instanciação, que institui a classe a criar um objeto que é um elemento ou instância da classe receptora.

5.2 Propriedades do Modelo Orientado a Objeto

- **Encapsulamento:** Cada objeto é visto como o encapsulamento de seu estado interno, suas mensagens, e seus métodos. A estrutura do estado e os pares mensagem-método são todos definidos pela classe à qual o objeto pertence. O valor do estado interno é determinado pelos métodos que o objeto executa em resposta às mensagens recebidas.

- **Polimorfismo:** É a propriedade que permite que uma mesma mensagem seja enviada a diferentes objetos, sendo que cada objeto executa a operação apropriada à sua classe. Mais importante: o objeto que envia a mensagem não precisa conhecer a classe do objeto receptor ou como aquele objeto irá responder à mensagem. Isto significa que a mensagem, por exemplo, "print" pode ser enviada a um objeto, sem a preocupação se aquele objeto é um caracter, um inteiro, uma string ou uma figura. O objeto receptor irá responder com o método que for apropriado à sua classe.

- **Herança:** É uma das propriedades mais importantes do modelo OO. Ela é possível via a definição de hierarquia de classes, isto é, uma estrutura em árvore de classes, onde cada classe tem 0 ou mais subclasses. Uma subclasse herda todos os componentes de sua classe-pai, incluindo estrutura do estado interno e pares método-mensagem. Toda propriedade herdada pode ser redefinida na subclasse, sobrepondo-se à definição herdada. Esta propriedade encoraja a definição de novas classes, sem duplicação de código.

5.3 Exemplo em C++

C++ é uma LP híbrida: possui todos os recursos de C mais alguns recursos de OO

Vantagens:

- uso de diferentes modelos para diferentes partes do problema
- reaproveitamento de código imperativo

C++ é um superconjunto de C. Módulos de C e C++ podem chamar uma ao outro.

5.3.1 Componentes de C++:

5.3.1.1 Classes

Classe em C++ é uma extensão de estruturas (records) de C, somando-se a habilidade de se definir componente da estrutura que são funções. Logo uma definição de classe consiste de uma coleção de declarações de variáveis e funções. Cada declaração pode ser pública ou privada, seguindo o modelo de TAD: elementos públicos são visíveis fora da classe; privados não são.

Forma geral de definição de classe:

```
class <nome_classe>{
   <decl. privadas>
public:
   <decl. públicas>
}
```

Exemplo:

```
class intervalo {
   float esq, dir;   /* variáveis privadas da classe */
public:
/* construtor da classe */
   intervalo(float e, float d)
      esq := e;
      dir := d;
```

```
    float esquerdo() return esq;
    float direito() return dir;
    int contem(float f)
        return (f esq) && (f < dir);
    int contemint (intervalo x)
        return (esq < x.esquerdo()) && (dir x.direito());
}
```

5.3.1.2 Objetos

São instâncias de Classes. São criados pela declaração de pertinência a uma classe.

Exemplo: **intervalo x;**

O objeto criado, x, tem suas próprias variáveis privadas - esq e dir - bem como suas próprias funções - esquerdo, direito, contem e contemint. O construtor - intervalo - também pertence ao objeto.

A função intervalo (o construtor) é chamada para todo objeto que é declarado com dois parâmetros. Por exemplo, x poderia ser inicializado como objeto intervalo (1.5, 2.8) pela declaração

intervalo x(1.5, 2.8);

Uma vez executada a declaração de um objeto, qualquer um de seus componentes públicos podem ser referidos, colocando-se seu nome como prefixo do objeto.

Exemplo: x.esquerdo(); /* refere-se ao limite do intervalo x à esquerda */

Isso significa uma chamada à função esquerdo, associada com ao objeto x. Os elementos na área privada do objeto são inacessíveis aos módulos externos. No entanto, é possível permitir o acesso de uma função externa a elementos privados de uma classe, através da declaração dessa função como friend function, dentro da classe. Isto não é muito recomendado pois representa uma violação ao Encapsulamento.

Exemplo:

```
class intervalo {
    friend void printintervalo(intervalo);
    ...
}
/* fora da classe intervalo */
void printintervalo(intervalo x) {
    printf("(%f, %f)", x.esq, x.dir);
}
```

Todas as funções de uma classe podem ser declaradas friends de uma dada classe, se a classe for declarada como friend.

5.3.1.3 Herança

É implementada em C++ via o conceito de classe derivada. Uma classe derivada de uma classe-base herda todos os membros públicos da classe-base. Assim, a classe derivada é a sub-classe, e a classe-base é a superclasse.

A declaração de uma classe derivada é da forma:

```
class <nome_classe_derivada>: public <nome_classe_base> {
   <declarações classe derivada>
};
```

Isso faz todos os membros públicos da classe base serem também membros públicos da classe derivada.

Uma variação de **Herança Publica** é a **Herança Privada** que é declarada como:

```
class <nome_classe_derivada> : <nome_classe_base> {
   <declarações classe derivada>
};
```

Nesse caso, os membros públicos da classe base são herdados como privados da classe derivada. Note que os membros privados da classe base não podem ser herdados.

5.3.1.4 Funções Virtuais

Se uma função numa classe_base é declarada virtual, então ela pode ser redefinida numa classe derivada.

Exemplo:

```
class top {
   public:
      virtual print1() { printf("top1\n"); };
              print2()  { printf("top2\n"); };
              print3()  { printf("top3\n"); };
              print()   { print1(); print2(); print3();} ;
};
class bottom: public top {
   public:   /* herda print3 e print e redefine as demais */
      print1() { printf("bottom1\n"); };
      print2() { printf("bottom2\n"); };
};
void main( void ) {
   top t;      bottom b;
   t.print();  b.print();
}
--------- Saída ----------
top1
top2
top3
bottom1
top2   // print2 não foi declarada virtual em top
top3
```

5.3.1.5 Overloading

C++ permite o overloading de operadores, funções e construtores.

Exemplo: classe amount

```
class amount {
        int tc;
    public:
        amount(int d, int c) { tc = 100 * d + c;};
        amount() {tc = 0;}

        amount operator* (int v)    { return amount(tc*v /100, tc*v % 100); };
        amount operator* (float v) { int newtc = v*tc + 0.5;
                                     return amount(newtc / 100, newtc % 100);  ;
        amount operator+ (amount a) { int newtc = a.tc + tc;
                                     return amount(newtc / 100, newtc % 100);} ;
        amount operator - (amount a) { int newtc = a.tc - tc;
                                     return amount(newtc / 100, newtc % 100);};
        int operator > (amount a) { return tc > a.tc; } ;

        int cents() { return tc % 100; };
        int dollars() { return tc / 100; };
        void setamount(float f) { tc = f * 100.0; };
        void setamount(int d, int c) { tc = 100 * d + c; };
        void printamount() {
            printf("%d.", tc / 100);
            if (tc % 100 < 10) {printf("0");}
            printf("%d\n", tc % 100);
        }
        int totalcents() { return tc; };
}
```

Dois construtores: um sem parâmetros, outro com dois parâmetros. Assim, um objeto pode ser declarado de duas maneiras:

```
#include "amount.h"

amount a(5, 25);  /* inicializa a com 5 dólares e 25 cents */
amount b;         /* inicializa b com 0.00 */
```

Exemplo: classe Account

```
class account {
  public:
        amount balance;
        char monthlyreport[1000];
        char name[30];
        char acctD[10];

        void addline(char* mr, char* type, char* id, amount a, char* date,
                     amount bal) ;
        void topline();


        account(char* ID, char* custname)  { numofaccts++;
            balance.setamount(0, 0);
```

```
                    *monthlyreport = 0; strcpy(name, custname);
                     strcpy(acctID, ID);
                     topline();
                }

                void deposit(char* itemID, amount amt, char* date)  {
                     totbal = totbal + amt;
                     balance = balance + amt;
                     addline(monthlyreport, "DEP", itemID, amt, date, balance);
                }
                void withdraw(char* itemID, amount amt, char* date)  {
                     if ( ! amt.balance )  {
                        balance = balance - amt;
                        totbal = totbal - amt;
                       addline(monthlyreport, "WTH", itemID, amt, date, balance);
                 }
                 void monthend() {
                     printf(monthlyreport);
                     *monthyreport = 0;
                 }
          }
```

Exemplo: declarações e chamadas de funções para classe Account e Amount:

```
account x("x1001", "Joao Silva");
amount a;
a.setamount(100,0);
x.deposit("011", a, "01/01/95");
```

Exemplo: Classes derivadas: checking e savings

```
#include <amount.h>
#include <account.h>
amount monservchg(5, 0);
float intrate = 6.0;

class checking : public account {
public:
   void check(char* itemID, amount amt, char* date)  {
        withdraw(itemID, amt, date);
   };
   void monthend(char* date)  {
        withdraw("SVC", monservchg, date);
        account::monthend();
    };
}
class savings : public account {
  public:
      void monthend(char* date) {
         deposit("INT", balance*(intrate/1200.0), date);
         account::monthend();
      };
}
```

5.4 Paradigma Orientação a Objeto x Paradigma Imperativo

O conceito de classe no modelo OO é essencialmente idêntico ao de TAD (Tipo Abstrato de Dados). Ambos apresentam uma definição encapsulada de estruturas de dados e procedimentos, permitem esconder informações, e são instanciados em objetos.

A maior diferença está no fato de que classe provê polimorfismo e herança.

Polimorfismo em Linguagens Imperativas: em ADA existe o overloading de procedures ou operadores.

Herança em Linguagens Imperativas: em Pascal existem subtipos, que são limitados a tipos escalares.