Universidade Federal do Piauí
Centro de Educação Aberta e a Distância

LINGUAGENS DE PROGRAMAÇÃO

Francisco Vieira de Sousa

Ministério da Educação - MEC
Universidade Aberta do Brasil - UAB
Universidade Federal do Piauí - UFPI
Centro de Educação Aberta e a Distância - CEAD

Linguagens de Programação

Francisco Vieira de Sousa

UFPI
2013

S729l Souza, Francisco Vieira de
Linguagens de Programação/Francisco Vieira de Souza. – Teresina: UFPI/UAPI. 2009.
120p.

Inclui bibliografia
1– Características das Linguagens. 2 – Variáveis e Tipos. 3 – Controle do Fluxo de Execução. 4 – Programação Orientada a Objetos. I.
Universidade Federal do Piauí/Universidade Aberta do Piauí. II. Título.

CDD: 511.3

Apresentação

Este texto é destinado aos estudantes do programa de Educação a Distância da Universidade Aberta do Piauí (UAPI), vinculada ao consórcio formado pela Universidade Federal do Piauí (UFPI); Universidade Estadual do Piauí (UESPI); Instituto Federal de Educação, Ciência e Tecnologia do Piauí (IFPI); com apoio do Governo do estado do Piauí, através da Secretaria de Educação. O texto é composto de quatro unidades, contendo itens e subitens que discorrem sobre as Linguagens de Programação, evidenciando como estas estruturas podem ser utilizadas no estudo da Informática.

Na **Unidade 1**, são estudadas as principais características que uma linguagem de programação deve apresentar para que seja considerada uma boa linguagem. A unidade tem início com a justificativa da necessidade de se estudar as Linguagens de Programação e termina com uma discussão sobre os principais paradigmas atualmente considerados.

Na **Unidade 2**, são analisadas as variáveis e os tipos de dados, buscando entender os diversos atributos que as variáveis têm, envolvendo a vinculação que é a associação entre as variáveis e seus atributos. Os tipos de dados também são analisados à luz da notação matemática, já de domínio público.

Na **Unidade 3**, se analisa o controle do fluxo de execução dos programas nos níveis: de expressão, instrução e de subprogramas, finalizando com um estudo sobre corrotinas.

Na **Unidade 4**, se analisam os tipos abstratos de dados e como eles podem ser implementados em Ada, C++ e Java. Em seguida, é feito um estudo sobre a Programação Orientada a Objetos, tanto do ponto de vista teórico, quanto prático.

Sumário

11 **UNIDADE 1**
CARACTERÍSTICAS DAS LINGUAGENS

Introdução 11
Por que estudar LP? 11
Características das linguagens de programação 12
O desenvolvimento de software 12
O processo de tradução das linguagens 16
Análises léxica e sintática das LPs 23
Paradigmas das linguagens de programação 27

35 **UNIDADE 2**
VARIÁVEIS E TIPOS DE DADOS

Introdução 35
Atributos de uma variável 36
Tipos de dados 46
Equivalência de tipos 57

35 **UNIDADE 3**
CONTROLE DE FLUXO

Introdução 65
Controle de execução em nível de expressões 65
Controle de fluxo em nível de instruções 72
Controle do fluxo em nível de subprogramas 81

96

UNIDADE 4
ABSTRAÇÃO DE DADOS

Introdução 97
O conceito de abstração 97
O conceito de encapsulamento 98
Tipos de dados abstratos definidos pelo usuáro 100
Progamação orientada a objetos 106
Herança 107
Poliformismo 110
Classes Abstratas 110

REFERÊNCIAS 115

UNIDADE 1

Características das Linguagens

Resumindo

O objetivo principal desta Unidade é apresentar os conceitos básicos das Linguagens de Programação, buscando capacitar o aluno na avaliação e comparação de LPs. É comum ver em fóruns de programação debates de desenvolvedores defendendo sua linguagem preferida, levantando algumas características da linguagem, outras vezes, apenas dizendo que a LP é melhor, sem especificar suas vantagens e desvantagens, sem apresentar as propriedades desejáveis em uma LP. Imaginamos que em um mundo global competitivo, não devemos nem podemos nos limitar a uma única linguagem de programação. É necessário ampliar o leque de opções, aprendendo as características básicas e os conceitos em que elas são implementadas. As características mudam muito entre as LPs e estaria relacionado, por exemplo, a sua sintaxe e recursos e isso muda rapidamente, levando a um estudo constante por parte dos desenvolvedores. Já os conceitos e metodologia se modificam a uma velocidade bem menor. Logo o profissional que tenha uma maior habilidade em aprender rapidamente uma nova linguagem possui maiores chances de sucesso nesta área.

1 CARACTERÍSTICAS DAS LINGUAGENS

INTRODUÇÃO

As LPs (Linguagens de Programação) são destinadas a serem usadas pelas pessoas para expressar um processo através do qual um computador pode resolver um problema. É a forma de comunicação entre as pessoas e o computador, quando elas desejam que ele execute determinadas tarefas, necessárias para a solução de determinados problemas.

POR QUE ESTUDAR LP?

Existem vários motivos que exigem que os estudantes de Computação conheçam, com profundidade, as características e princípios que as linguagens de programação devem apresentar e como são implementadas. Entre eles, podem ser citados:

1. *Maior habilidade em resolver problemas:* uma maior compreensão de uma LP pode aumentar nossa habilidade em pensar em como atacar os problemas, tanto melhor se dominarmos os vários modelos de LPs.
2. *Melhor uso de uma LP*: a compreensão das funções e implementação das estruturas de uma LP, nos levam a usá-la de forma a extrair o máximo de sua funcionalidade e eficiência.
3. *Melhor escolha de uma LP*: o conhecimento das características das linguagens permite aos profissionais da área de software, escolher a que seja adequada à solução desejada para um determinado tipo de problema.
4. *Maior facilidade em aprender novas LPs*: o conhecimento dos

conceitos chaves comuns às LPs provoca maior facilidade no aprendizado de novas linguagens.

5. *Melhor projeto de LPs:* o conhecimento de novas linguagens de interfaces de sistemas, extensão de LP via operadores e tipos de dados, permite melhores projetos de novas linguagens.

CARACTERÍSTICAS DAS LINGUAGENS DE PROGRAMAÇÃO

As características desejáveis em uma linguagem de programação têm muito a ver com a aplicação que se tem em mãos. No entanto, as linguagens devem obedecer a alguns critérios que devem ser comuns a todas elas. O conhecimento destes critérios e propriedades é o objetivo principal deste estudo e é o tema desta seção.

Legibilidade

Esta característica está relacionada com a facilidade com que um programador lê e entende rapidamente o código de um programa. Quanto mais fácil for ler um programa, mais fácil será entender o código e também descobrir erros na programação. Uma LP com baixa legibilidade torna o seu aprendizado mais difícil. Vejamos alguns exemplos:

- Linguagens que usam Goto normalmente reduzem a legibilidade porque nesse tipo de programação, os programas possuem fluxo de controle e não obedecem a padrões regulares. Tornando difícil acompanhar e entender o que eles fazem.
- Uso de mesmo vocábulo da LP para denotar diferentes comportamentos dependendo do contexto é prejudicial à legibilidade e entendimento da LP.

Ex: VISUAL BASIC:

- Uso do operador “=” com sinal de atribuição e comparação.
- Chamada funções e procedimentos com passagem de parâmetros. De acordo com o contexto é necessário o uso de “(“.

Ex: C/C++:

- Uso do “*” denota diversas coisas, como sinal multiplicador, passagem por referência, declaração de ponteiros.

- Efeitos colaterais são prejudicais à legibilidade. Eles causam mudanças adicionais no estado do programa durante a avaliação de uma determinada expressão ou a execução de um comando ou subprograma.

Ex: VISUAL BASIC:

- A não necessidade de informar os parâmetros aos procedimentos e funções, passagem por valor ou referência. Adotando como padrão passagem por referência, podendo ocasionar efeito colateral indesejável.

Ex: Marcadores de blocos de comandos como o “begin end” (Pascal e Delphi) e o “{ }” de C/C++ e JAVA, também podem causar confusões na leitura do programa quando existem vários comandos de repetição e seleção aninhados. Além disso, a não obrigatoriedade de usar um marcador específico para indicar onde o comando “**if**” do C se encerra, possibilita a escrita de comando “ifs” aninhados, difíceis de serem entendidos.

Ex: Algumas LP’s adotaram postura altamente questionável com relação à legibilidade. FORTRAM, por exemplo, permite que identificadores especiais como DO, END, INTEGER e REAL sejam também nomes de variáveis.

Redigibilidade

A redigibilidade está relacionada com a facilidade com que um programador codifica programas. A redigibilidade de programas pode conflitar com a legibilidade. Por exemplo, a linguagem C permite a codificação de comandos complexos, mas podem não identificar de maneira muito clara a sua funcionalidade.

As LPs com tipos de dados limitados requerem o uso de estruturas complexas, o que acaba dificultando a codificação de programas. A falta de declaração recursiva e ponteiro em Visual Basic acaba limitando o seu uso para implementar programas com uso de estruturas de árvores, listas e etc.

Outro ponto de falta de redigibilidade em Visual Basic está relacionado com a declaração de variáveis, onde não é possível declarar várias variáveis do mesmo tipo, especificando o tipo somente uma vez.

Confiabilidade

Essa propriedade se relaciona Aos mecanismos fornecidos pela LP para incentivar a construção de programas confiáveis. LPs que requerem a declaração de dados permitem verificar, automaticamente, erros de tipos durante a compilação ou execução. LPs que possuem mecanismos para detectar eventos indesejáveis (Tratamentos de Erros) e especificar respostas adequadas a tais eventos permitem construção de programas mais confiáveis.

Eficiência

De acordo com as demandas por recursos de um tipo de aplicação, certas LPs são mais recomendadas, e outras não devem ser usadas. Aplicação de automação em tempo real, por exemplo, normalmente requerem o uso de LPs que minimizem o tempo de execução e de acesso aos dispositivos periféricos, bem como o consumo de especo de memória.

Por exemplo, PASCAL, JAVA, VISUAL BASIC exigem que os índices de vetores sejam verificados em todos os acessos durante a execução dos programas. Isso implica na necessidade de fazer um teste antes de qualquer acesso aos vetores. Por outro lado, como o C não exige este tipo de teste, o código gerado tem um desempenho melhor.

Em contrapartida, o fato de C não verificar os índices de vetores provoca alguma perda na confiabilidade de programas codificados nela. Pode acarretar acesso a espaço de memória erradamente, tornando a execução do programa imprevisível. É comum a quem programa em C, testar o programa com sucesso, e, ao desligar a máquina e testar novamente, ocorrer erros. Isso acontece devido à coincidência do acesso à memória, de forma errada. Na realidade, C é uma linguagem para desenvolvedores que “sabem” o que estão fazendo.

Facilidade de aprendizado

O programador deve ser capaz de aprender a linguagem com facilidade. LPs com muitas características e múltiplas maneiras de realizar a mesma funcionalidade, tendem a ser mais difíceis de aprender. Além disso, outro aspecto negativo causado pelo excesso de características é o fato de levar os programadores a conhecerem apenas uma parte da linguagem, o

que torna mais difícil a um programador entender o código produzido por outro.

Ortogonalidade

Este termo diz respeito à capacidade da LP permitir ao programador combinar seus conceitos básicos, sem que se produzam efeitos anômalos nessa combinação. Assim, uma LP é tão mais ortogonal quanto menor for o numero de exceções aos seus padrões regulares.

LPs ortogonais são interessantes porque o programador pode prever, com segurança, o comportamento de uma determinada combinação de conceitos. Isso pode ser feito sem que se tenha de implementar testes para a averiguação do uso combinado de dois ou mais conceitos, ou mesmo, buscar na especificação da LP se existe alguma restrição àquela combinação.

A falta de ortogonalidade diminuiu o aprendizado da LP e pode estimular a ocorrência de erros de programação.

Ex: A não necessidade de declaração de variáveis no Visual Basic (default tipo variant); não permitir declaração de várias variáveis de mesmo tipo, na mesma linha (default por referência); permitir não especificar passagem de valor ou referência; permitir não especificar o escopo de procedimentos e funções (default public).

Reusabilidade

A reusabilidade é a possibilidade de reutilização do mesmo código para diversas aplicações. Quanto mais reusável for um código, maior será a produtividade de programação, uma vez que, na construção de novos programas, bastará utilizar e, eventualmente, adaptar códigos escritos anteriormente sem que se faça necessário reconstruí-los. A grande maioria das LPs permite reuso de código através da modularização por meio das bibliotecas de subprogramas. Por exemplo, em Visual Basic, permite ao programador modularizar o seu código por módulos ou classes e ainda permite programação de dll e criação de componentes.

Modificabilidade

Refere-se às facilidades oferecidas pela LP para possibilitar ao

programador alterar o programa em função de novos requisitos, sem que tais modificações impliquem mudanças em outras partes do programa. Exemplo de mecanismos que proporcionam boa modificabilidade são o uso de constantes simbólicas e a separação entre interface e implementação na construção de subprogramas e tipos de dados abstratos.

Portabilidade

É altamente desejável que programas escritos em uma LP se comportem da mesma maneira, independentes da ferramenta utilizada para traduzi-los para a linguagem de máquina ou da arquitetura computacional (hardware ou sistema operacional) sobre a qual estão sendo executados.

Dessa maneira, um mesmo programa ou biblioteca pode ser utilizado em vários ambientes e diferentes situações, sem que seja necessário despender tempo de programação para reescrevê-los ou adaptá-los ao novo ambiente de tradução ou execução.

O DESENVOLVIMENTO DE SOFTWARE

Muitas mudanças, algumas radicais, aconteceram de forma rápida na história da computação, e, com o barateamento dos custos dos equipamentos de computação, grande parte da importância inicialmente atribuída ao *hardware* passou para o *software*. Máquinas mais complexas e poderosas exigem programação mais sofisticada, para que o seu potencial possa ser aproveitado, e a produção de software é hoje fundamental para virtualmente qualquer espécie de atividade humana.

O *"ciclo de vida"* a seguir é um dos diversos modelos que nos auxiliam a compreender essa complexa atividade, e descreve, de forma simplificada, as fases da história de um programa:

- *Especificação de requisitos:* nesta fase se procura definir com clareza qual é o problema considerado, e que características deve ter uma solução computacional (um sistema de hardware e software adequados), para que essa solução seja satisfatória para a aplicação e o usuário.
- *Projeto global:* nesta fase, procura-se definir, em linhas gerais, como deve ser o software a ser construído, para estabelecer a viabilidade de sua construção e de sua utilização nas condições

reais. É nesta fase que deve ser escolhida a linguagem (ou possivelmente as linguagens) de programação a utilizar.

- *Codificação*: Nesta fase, o software é efetivamente escrito em uma ou mais linguagens escolhidas para isso.
- *Validação*: nesta fase, o sistema é testado, ou de alguma outra forma examinado, para que se tenha uma garantia de seu funcionamento satisfatório.
- *Utilização e evolução*: nesta fase, o sistema finalmente preenche seus objetivos. Se existem, são então descobertos erros e problemas que, de alguma forma, tornam o sistema inadequado para o uso pretendido. São também examinadas possibilidades de alteração e extensão do sistema que visam torná-lo mais adequado para a aplicação, cujas necessidades já podem ser diferentes daquelas consideradas na fase de especificação de requisitos. As alterações a serem feitas constituem o que se chama de manutenção ou evolução do software.

Naturalmente, cada uma das fases mencionadas pode ser várias vezes repetidas, até que o produto final seja considerado satisfatório. Quanto mais cedo um erro for descoberto, mais simples e barata é sua correção, e por essa razão, é importante que os erros sejam descobertos o mais cedo possível, nas fases iniciais do ciclo de vida.

Note-se que a escolha da linguagem de programação influi, decisivamente, apenas em duas das fases do ciclo apresentado: a fase de codificação, onde o programa é escrito na linguagem, e a fase de evolução/manutenção, em que o programa é alterado, através da introdução ou substituição de trechos de código escritos na linguagem. Nestas fases, portanto, é que defeitos e qualidades de uma linguagem de programação se tornam aparentes.

Naturalmente, uma linguagem usada para programação de protótipos não precisa exatamente das mesmas qualidades de uma linguagem de programação de uso geral. Em alguns casos, protótipos podem ser construídos em linguagens como Prolog ou Lisp, linguagens para as quais legibilidade e eficiência não foram características dominantes de projeto; posteriormente, se necessário, versões definitivas podem ser construídas em outras linguagens mais adequadas.

Outras qualidades a serem consideradas em linguagens de programação estão listadas a seguir.

- **Eficiência do programa compilado:** a estrutura da linguagem facilita 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.
- **Eficiência do processo de compilação:** a estrutura da linguagem facilita o processo de compilação, permitindo a construção de compiladores e ambientes de programação em que seja mais eficiente o processo de desenvolvimento de software na linguagem.
- **Disponibilidade de ferramentas:** existem compiladores, interpretadores, ambientes de programação e de desenvolvimento amigáveis, adequados para a linguagem nas máquinas em que a implementação deve ser realizada. Além disso, essas ferramentas devem poder ser adaptadas a casos particulares, permitindo a máxima eficiência em cada caso. Por exemplo, os compiladores devem ter opções de compilação que permitam gerar o melhor tipo de código objeto para cada aplicação.
- **Disponibilidade de bibliotecas:** existem bibliotecas de software de uso geral e de uso específico para a aplicação, que permitem a utilização de software já escrito, dispensando a necessidade de sua construção. Essas bibliotecas podem ser de texto fonte na linguagem, ou de código pré-compilado, escrito em qualquer linguagem, mas que possa ser ligado ao código escrito na linguagem.

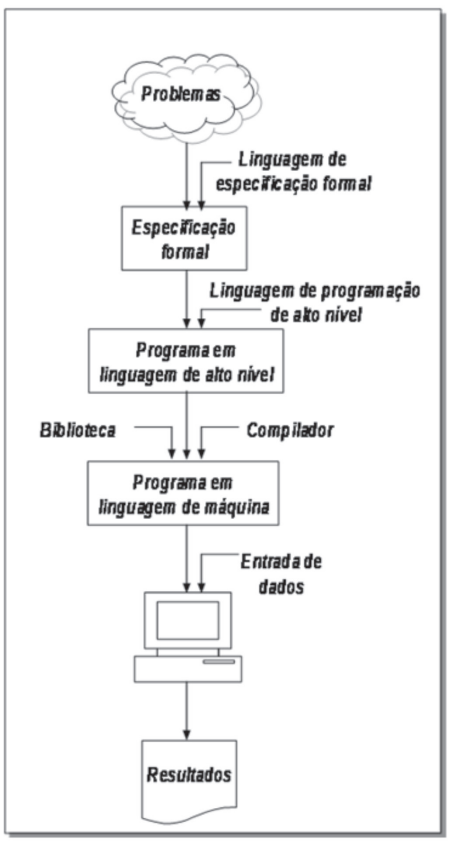


Figura 1 3. O processo de solução de um problema.

O processo de solução de um problema utilizando um programa de computador pode ser pensado como o gráfico mostrado na Figura 1.3 .

Na figura, é mostrado o problema de forma ainda confusa e que deve ser analisado através de

uma linguagem de especificação formal, gerando uma especificação que vai ser implementada em alguma linguagem de programação.

Existem muitas linguagens de especificação formal. Entre elas, podem ser citadas VDM, Lotos, Z, Redes de Petri e outras, dependendo da área em que o problema a ser resolvido se insere. A especificação formal vai ser implementada em alguma linguagem de programação. A escolha desta linguagem depende da experiência do programador e do conhecimento que ele tem sobre a adequação das linguagens a cada tipo de problema. Deve ser salientado, no entanto, que esta solução representa uma solução ideal e ainda é um processo em evolução e difícil de ser bem realizado.

O objetivo principal de um programa de computador é que ele seja correto, ou seja, que faça apenas a tarefa para a qual foi projetado e de forma correta. Esta exigência tem se tornado cada vez mais presente, à medida que os problemas se tornam, a cada dia, mais complexos e difíceis de serem resolvidos. Lembremos que o homem, atualmente, controla reatores nucleares e naves interplanetárias, e que cirurgias de alto risco podem ser realizadas por robôs. Isto significa que, inicialmente, os programas devem ser corretos e depois apresentar um bom desempenho.

Enquanto o desenvolvimento de provas matemáticas da corretude de programas não for completamente dominado, uma solução alternativa tem sido a submissão do programa a testes de validade. Esta técnica tem sido empregada com relativo sucesso, para programas que não tenham grande exigência quanto à necessidade de sua corretude. A escolha de bons testes tem sido uma área importante de pesquisa, lembrando que, os testes não provam, matematicamente, a ausência de erros; eles acusam a presença deles para determinados casos, mas não para todos os casos possíveis de entrada de dados.

O PROCESSO DE TRADUÇÃO DAS LINGUAGENS

Uma linguagem de programação pode ser convertida ou traduzida em código de máquina, por compilação ou interpretação. Ambas representando algum processo de tradução.

O processo de compilação

A compilação pode ser realizada de forma pura, onde o programa

é totalmente analisado e traduzido por outro programa conhecido como *compilador*, para um programa em linguagem de máquina. O compilador é constituído pelos analisadores léxico, sintático e semântico, que utiliza uma tabela de símbolos para realizar o processo de tradução. Esta tabela é frequentemente consultada pelos analisadores durante o processo de tradução. Os analisadores léxicos e sintáticos serão vistos, com um pouco mais de detalhes, ainda nesta unidade. O programa traduzido em linguagem de máquina pode ser armazenado em alguma mídia, e ser executado em outra oportunidade, e possivelmente, por outra máquina. Esse processo de compilação pura está mostrado graficamente na Figura 1.4 a seguir.

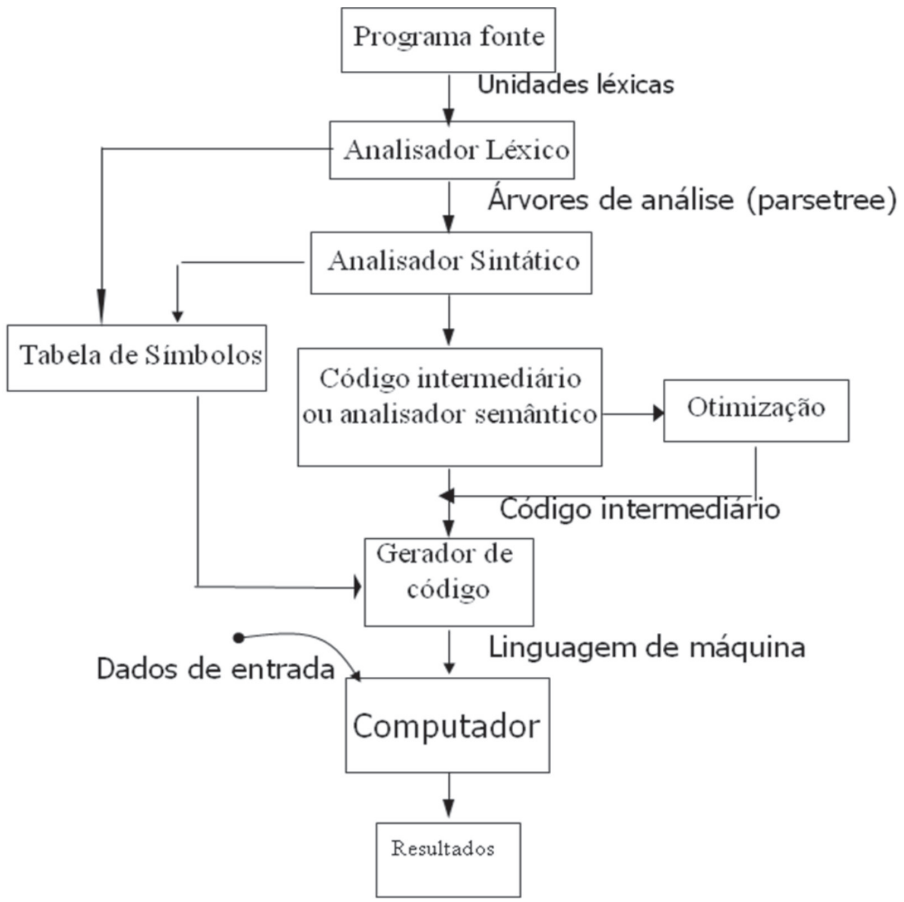


Figura 1.4. Processo de compilação de um programa.

O processo de interpretação pura

Por outro lado, se o texto do programa for sendo traduzido, à medida que vai sendo executado num processo de tradução de trechos seguidos de sua execução imediata, então, diz-se que o programa foi *interpretado* e que o mecanismo utilizado para a tradução é um interpretador. Programas interpretados são, geralmente, mais lentos que os compilados, mas são também mais flexíveis, já que podem interagir com o ambiente, mais facilmente. Esse processo, conhecido como *interpretação pura*, está mostrado na Figura 1.5, a seguir.

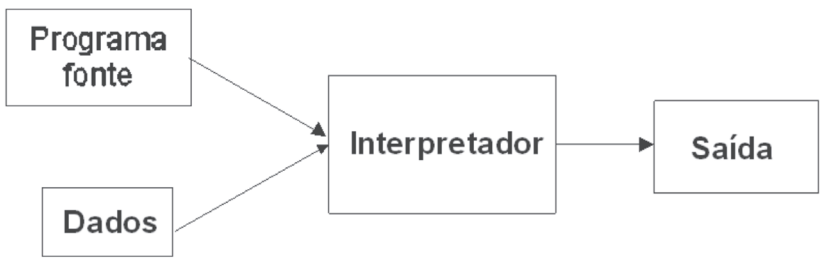


Figura 1.5. Processo de interpretação pura

O processo de interpretação híbrida

Embora haja essa distinção entre interpretação e compilação, as coisas nem sempre são tão simples como parecem. Existem linguagens onde o processo de implementação é uma mistura entre os dois métodos, ou seja, há uma tradução inicial para um código intermediário e este código é executado por um interpretador. Este método híbrido tem a vantagem de que, no processo de tradução do código fonte para a representação intermediária, os erros sintáticos são detectados e corrigidos, fazendo com que a representação intermediária seja um código isento destes tipos de erros.

A linguagem Java utiliza atualmente esta metodologia. Um compilador traduz o código em Java para o código intermediário (e portável) da JVM (Java Virtual Machine), conhecido como *bytecode*. Os *bytecodes* são executados nos diversos *browsers* ou navegadores através de seus interpretadores.

De forma gráfica, esse procedimento está mostrado na Figura 1.6, a seguir.

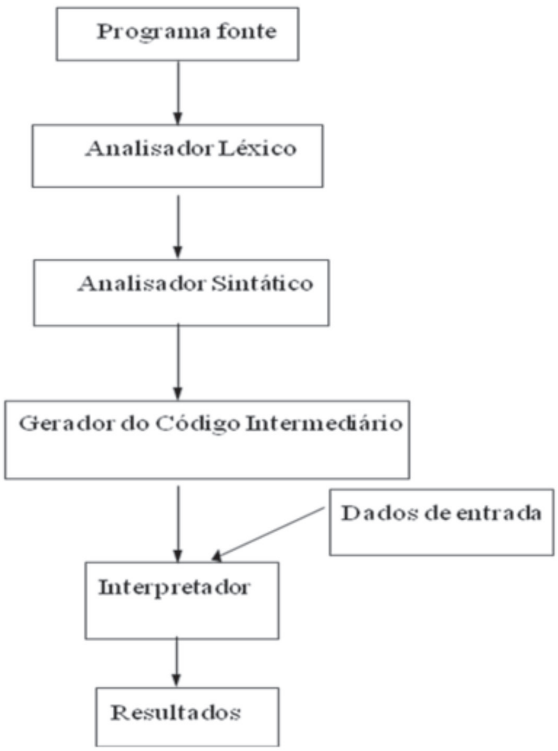


Figura 1 6. O processo de interpretação híbrida

Existem também outras formas de interpretar, em que os códigos-fontes, ao invés de serem interpretados linha-a-linha, têm blocos "compilados" para a memória, de acordo com as necessidades, o que aumenta o desempenho dos programas quando os mesmos módulos são chamados várias vezes. Essa técnica é conhecida como ***JIT*.**

As JVMs originais interpretavam esse código de acordo com o código de máquina do computador hospedeiro. Porém, atualmente, elas compilam segundo a técnica ***JIT*** o código JVM para código hospedeiro.

Deve ser salientado, no entanto, que as diferenças entre interpretadores e compiladores não podem ser reconhecidas externamente com facilidade. Em geral, o interpretador é mais lento, uma vez que alguns passos a serem realizados durante a execução de um programa, notadamente, a busca de instruções e a análise de cada uma destas instruções são repetidas cada

vez que uma nova instrução for executada; no caso do compilador, isso não é necessário. Adicionalmente, é possível adaptar o código gerado por um 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 interpretador 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.

ANÁLISE LÉXICA E SINTÁTICA DAS LPS

Na seção anterior, o leitor deve ter percebido nos gráficos que mostram os processos de compilação pura e de interpretação híbrida a presença dos analisadores léxico e sintático, que recebem como entrada o programa codificado em alguma linguagem de alto nível, como C, Pascal, Java e outras.

Para entender este processo, vamos, inicialmente, analisar como o mesmo é feito nas linguagens naturais, como o Português. Por exemplo, podemos escrever a oração “nós é bacana”, onde verificamos um erro de concordância porque o verbo não está concordando com o sujeito.

Algo semelhante existe na verificação da corretude de um programa computacional, onde a linguagem utilizada é artificial. Por exemplo, a instrução em C, **“if x = 10 y = 12”** apresenta alguns erros: o comando **if** exige que a expressão **booleana** de teste seja escrita entre parênteses, o operador de teste de igualdade é simbolizado por **“==”** e toda instrução em C termina com um ponto e vírgula. Isto significa que a instrução correta seria **“if (x == 10) y = 12; “**. A análise e detecção destes erros é o papel dos analisadores léxico e sintático, vistos nas figuras da seção anterior.

Análise léxica e sintática se refere à forma. Existe, no entanto, a análise semântica que se relaciona ao significado das instruções. Uma instrução pode estar sintaticamente correta, mas, semanticamente, errada. Em Português, temos algo semelhante. Por exemplo, podemos escrever “nós comemos o sol”. Sintaticamente está correta, mas não apresenta um significado coerente. Um fenômeno semelhante pode ocorrer nos programas computacionais, ou seja, uma instrução ou um conjunto delas pode estar correto sintaticamente e, no entanto, pode não apresentar um significado coerente. Este é o caso de uma análise semântica e que é bem mais

complicada de ser feita pelos compiladores. Resumidamente, os processos de análise léxica e sintática são bem mais fáceis de serem realizados pelos compiladores, que o processo de análise semântica.

Existem procedimentos formais para a construção de analisadores sintáticos e semânticos, mas, estes últimos, ainda carecem de muita pesquisa para serem construídos com facilidade.

A descrição formal de uma LP por BNF

Na década de 50, John Backus e Noam Chomsky criaram metodologias para descrever a sintaxe das linguagens de programação e das linguagens naturais, respectivamente. Chomsky dividiu as linguagens naturais em quatro classes de gramáticas. Onde, duas delas, ***as linguagens livres de contexto e as linguagens regulares***, foram utilizadas para descrever a sintaxe das linguagens de programação.

Peter Naur e John Backus construíram uma metodologia para descrever as linguagens de programação que ficou conhecida como Backus-Naur-Form, popularmente conhecida como BNF, que se tornou uma metalinguagem, ou seja, uma linguagem utilizada para descrever outras linguagens. A BNF foi utilizada pela primeira vez para descrever a sintaxe da linguagem Algol 58, apresentada por John Backus em uma conferência internacional em 1959. Até hoje, a BNF é a metodologia mais utilizada para descrever linguagens artificiais.

A BNF usa abstrações para estruturas sintáticas. Uma simples instrução de atribuição em C, por exemplo, poderia ser representada pela abstração <atribuição>, onde os colchetes angulados são utilizados para delimitar abstrações que são conhecidas como “não terminais”. Uma instrução de atribuição em C pode ser representado em BNF por:

<atribuição> → <var> = <expressão>

Onde, o símbolo à esquerda da seta é chamado de “lado esquerdo”, LE, é a abstração que está sendo definida pelo “lado direito”. Cada instrução deste tipo é conhecida como **“regra de produção”** ou apenas **“produção”**.

O analisador léxico divide o programa, um string, em unidades sintáticas chamadas lexemas, que incluem os identificadores, os literais, os operadores e as palavras especiais da linguagem. Pode-se dizer que um programa é uma sequência de lexemas, em vez de uma sequência de caracteres.

Os lexemas são agrupados em determinada categoria formando os ***tokens***, ou seja, conjuntos de lexemas de uma mesma categoria, podem ser compostos por apenas um lexema. Por exemplo, a instrução C, **"cont = 4 * num + 5;"** é assim analisada:

Lexema	Token
cont	identificador
=	Sinal_de_igualdade
4	Literal_inteiro
*	Operador_mult
num	identificador
+	Operador_soma
5	Literal_inteiro
;	Ponto_e_vírgula

A BNF é um dispositivo generativo utilizado para definir linguagens, onde as sentenças da linguagem são geradas por uma sequência de aplicações das regras, iniciando por um símbolo inicial. Para a BNF, os tokens são conhecidos como símbolos **não terminais** e os lexemas são **símbolos terminais**. Os símbolos terminais só podem aparecer do lado direito da produção, enquanto os não terminais podem aparecer dos dois lados.

Exemplo 1.1. Seja a seguinte gramática:

<atrib> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <termo>
| <termo>
<termo> → <termo> * <fator>
| <fator>
<fator> → (<expr>)
| <id>

Verifiquemos que a sentença A = B * (C + D) pertence à linguagem definida por esta gramática.

<atrib> → <id> = <expr>

- A = <expr>
- A = <termo>
- A = <termo> * <fator>
- A = <fator> * <fator>
- A = <id> * <fator>

- A = B * <fator>
- A = B * (<expr>)
- A = B * (<expr> + <termo>)
- A = B * (<termo> + <termo>)
- A = B * (<fator> + <termo>)
- A = B * (<id> + <termo>)
- A = B * (C + <termo>)
- A = B * (C + <fator>)
- A = B * (C + <id>)
- A = B * (C + D)

Desta forma, a sentença pode ser derivada a partir da gramática, ou seja, é uma sentença da linguagem definida pela gramática.

O sistema de derivação anterior pode ser mostrado, graficamente, através de uma árvore, chamada de **"árvore de análise"** ou **"parse tree"**, onde os nós internos representam os não terminais, e as folhas da árvore representam terminais. Para o Exemplo anterior, a parse tree correspondente à sentença A = B * (C + D) está mostrada na Figura 1.7. Na figura, pode-se verificar a sentença observando as folhas da árvore.

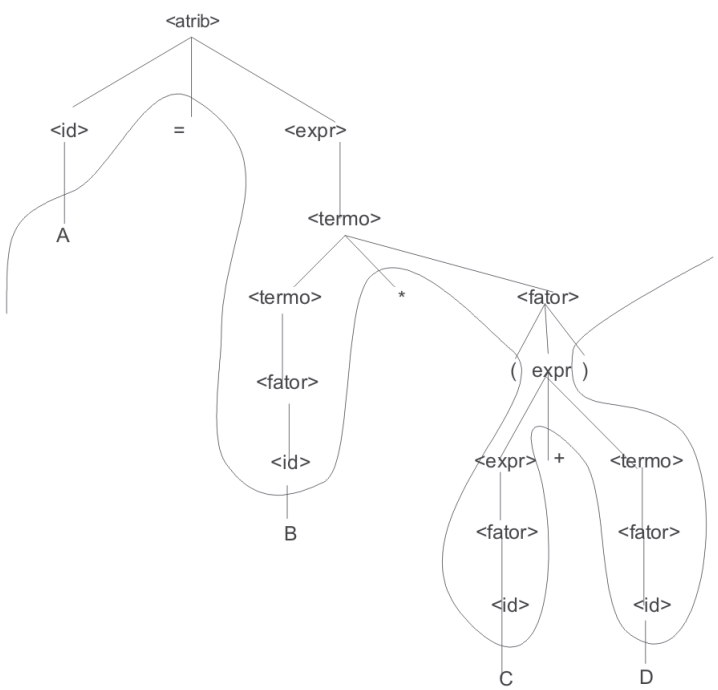


Figura 1.7 Parse tree para A = B * (C + D).

Algumas observações podem ser feitas sobre a parse tree. Inicialmente, muitas propriedades da gramática podem ser observadas diretamente na árvore. Por exemplo, verifica-se que a operação "+" da sentença aparece na árvore em um nível mais baixo que o do operador "*". Ao se implementar esta árvore, o processo deve ser iniciado pelas folhas, portanto, o operador que está mais abaixo na árvore vai ser implementado primeiro. Isto significa que este operador tem precedência sobre o anterior. De outra forma, a árvore mostra qual é a ordem de precedência entre os operadores da gramática.

Outra observação que pode ser feita diretamente da *parse tree* é que, se a árvore se expande pela direita ou pela esquerda, isto significa, que, a associatividade das operações é feita pela direita ou pela esquerda.

A *parse tree* é única para as gramáticas não ambíguas, ou seja, para cada sentença de uma gramática não ambígua, existe apenas uma *parse tree* associada a ela se a gramática for não ambígua.

PARADIGMAS DAS LINGUAGENS DE PROGRAMAÇÃO

As linguagens de programação são classificadas em paradigmas, algumas vezes chamados de modelos. Estes paradigmas são:

- *o paradigma imperativo;*
- *o paradigma funcional;*
- *o paradigma lógico e*
- *o paradigma orientado a objetos.*

Essa classificação não representa a unanimidade de pontos de vistas dos projetistas e pesquisadores das linguagens de programação. Sendo alegado, por alguns, que o paradigma orientado a objetos nada mais é que uma particularização do paradigma imperativo, muitos advogando a existência do paradigma concorrente, dado o advento das arquiteturas paralelas e a necessidade destas linguagens para permitir a programação de softwares para estas máquinas.

O paradigma imperativo

A grande maioria das linguagens obedece ao paradigma imperativo, sendo este, o modelo ao qual faremos referência neste estudo.

O modelo Imperativo é baseado na perspectiva do computador,

onde a execução sequencial de comandos e o uso de dados são conceitos baseados no modo como os computadores executam programas, no nível de linguagem de máquina. Este modelo é o predominante. As LPs imperativas são de fácil tradução. Um programa imperativo é equivalente a uma sequência de modificações de locações de memória.

Exemplos de linguagens Imperativas são: FORTRAN, COBOL, ALGOL 60, APL, BASIC, PL/I, SIMULA 67, ALGOL 68, PASCAL, C, MODULA 2, ADA, além de outras.

O paradigma funcional

O paradigma funcional apresenta uma metodologia diferente da imperativa, sendo baseada inteiramente em funções matemáticas, a sua base de projeto. Em uma programação funcional pura não existem atribuições destrutivas, ou seja, se for atribuído um valor a uma variável x, em algum ponto do programa, nenhum outro valor poderá ser atribuído mais a x, neste programa. Nos programas funcionais, as variáveis mais se parecem com constantes. Isto implica que programas não apresentam quaisquer efeitos colaterais. Os programas funcionais são altamente modulares e representam uma escolha importante para a execução paralela de programas. Até certo tempo atrás, as linguagens funcionais foram deixadas em um segundo plano, em parte, devido a algumas dificuldades que elas apresentavam em relação à computação com arquivos e entrada e saída. No entanto, este panorama tem se modificado com o surgimento de novas pesquisas e tecnologias, observando-se um aumento cada vez maior de sistemas codificados em linguagens funcionais, onde a sua forte fundamentação matemática torna estes programas bem mais fáceis de serem provadas as suas corretudes. Há de ser salientado que, um programa funcional tem, em média, um quinto do tamanho de um programa imperativo para resolver o mesmo problema. Em muitos casos, este valor chega a 1/20 avos. Além disso, muitas bibliotecas e ferramentas de ajuda ao desenvolvimento de sistemas têm sido projetadas.

Exemplos dessas linguagens podem ser Lisp, SML, Miranda, KRC, Erlang e Haskell, onde muitos sistemas têm sido codificados e em diversas áreas de aplicação.

O paradigma lógico

O paradigma lógico utiliza a Lógica simbólica como fundamento para sua programação, notadamente a Lógica de Predicados de 1ª ordem, portanto, muito diferente das linguagens imperativas e das funcionais. Para atingir seus objetivos, a programação lógica utiliza de regras de inferências e as cláusulas de Horn, além da teoria dos quantificadores universal e existencial da Lógica de Predicados. Este paradigma ainda não tem apresentado uma grande opção por parte dos programadores, principalmente, devido ao desempenho apresentado com o atual estágio de desenvolvimento do hardware e a dificuldade de se programar neste paradigma. Atualmente, a única linguagem que tem representado este paradigma tem sido Prolog.

O paradigma orientado a objetos

O modelo Orientado a Objeto focaliza mais o problema. Um programa OO é equivalente a objetos que mandam 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 Simulada, 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 aos outros.

Os componentes básicos de uma linguagem orientada a objetos são os seguintes:

- **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.

O paradigma orientado a objeto para ser operacional tem de apresentar as seguintes propriedades:

- **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 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 a 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.

EXERCÍCIO

1. Qual a utilidade do conhecimento sobre as linguagens de programação?

2. Que linguagem de programação dominou a computação científica ao longo dos anos?

3. Que linguagem de programação dominou a computação comercial ao longo dos anos?

4. Que linguagem de programação dominou a inteligência artificial ao longo dos anos?

5. O que significam legibilidade, redigibilidade e ortogonalidade, no contexto de características de linguagens de programação?

6. Quais foram as duas deficiências das linguagens de programação descobertas em consequência da pesquisa em desenvolvimento de software da década de 70?

7. Quais as diferenças existentes entre os programas interpretados e compilados?

8. O que significa interpretação híbrida no contexto de implementação de linguagens?

9. Quais os paradigmas das linguagens de programação e que áreas cada um se destaca?

Exercício. Usando a gramática do Exemplo 1.1 mostre uma árvore de análise e uma derivação das seguintes instruções:

a) A = (A + B) * C

b) A = B + C + A

c) A = A + (B * C)

d) A = B * (C * (A + B))

SAIBA MAIS

O material mostrado nesta unidade foi, em grande parte, retirado do livro do Robert Sebesta, que consideramos uma excelente referência neste assunto. As anotações de aula do Professor José Lucas M. Rangel Netto, da Puc-Rio, também apresentam um material de boa qualidade. No entanto, muito outras referências sobre este tema existem e podem ser consultadas pelo leitor, até porque, este é um tema recorrente para os profissionais da área da Informática e da Computação.

UNIDADE 2

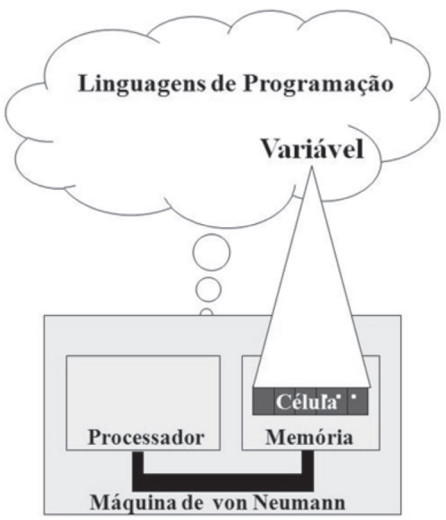
Variáveis e Tipos de Dados

Resumindo

O objetivo principal desta Unidade é conhecer com profundidade conceitos inerentes às variáveis e aos tipos de dados. Os conceitos e propriedades referentes a estes temas são importantes para entender o comportamento de programas. As variáveis e seus atributos decidem uma parte importante da computação e os tipos de dados são importantes na verificação da corretude dos tipos e das operações que estes tipos podem realizar com os valores das variáveis. Outra parte importante dos sistemas computacionais diz respeito ao gerenciamento da memória em tempo de execução. Este gerenciamento deve ser feito de forma automática pelo sistema em vez de ser realizado pelo programador que pode cometer algum erro ao esquecer uma desalocação necessária. As células que não são mais referenciadas em um programa devem ser incorporadas ao rol de células livres para que possam ser reutilizadas e compartilhadas. A forma de apresentação utilizada é de acordo com o exigido para o ensino a distância, ou seja, tendo em vista sempre esta nova modalidade de ensino.

2 VARIÁVEIS E TIPOS DE DADOS

INTRODUÇÃO



As linguagens de programação imperativas representam, em um nível mais abstrato, a arquitetura de von Neumann, onde a memória e o processador são os elementos básicos. A memória porque é onde são armazenadas as instruções de um programa em linguagem de máquina e também os dados necessários para a execução de um programa. Neste sentido, as células de memória são representadas pelas variáveis em um nível mais alto e, portanto, mais abstrato. Por exemplo, uma variável *cont* do tipo inteiro em um programa C, representa, em um nível mais abstrato, uma célula de memória composta de 32 ou 64 bits. Em um nível mais baixo, temos uma célula de 32 ou 64 bits, e uma representação desta célula em um nível mais abstrato é a variável que tem um nome identificador que é *cont*. Na realidade, o nome é apenas um dos vários atributos que as variáveis têm, uma vez que outros também existem e são objetos de estudo nesta unidade.

Entre os atributos que as variáveis têm, um se destaca, que é o *tipo* da variável. Dizer que uma variável é de um determinado tipo, significa que se está informando quais os possíveis valores que ela pode assumir e quais operações podem ser realizadas sobre os valores deste tipo, ou seja, com aquela variável. Deve ser notado que esta mesma observação se aplica, de forma idêntica, aos conjuntos matemáticos.

ATRIBUTOS DE UMA VARIÁVEL

Já foi observado que as variáveis apresentam algumas propriedades e características que são chamadas de atributos. São 6 (seis) os atributos principais das variáveis: o nome, o valor, o tipo, o endereço, o escopo e o tempo de vida. Estes atributos podem ser observados na figura a seguir.

Na figura aparece, em cor escura, uma outra entidade que é a **vinculação** (*binding*). Na realidade, a vinculação é uma associação entre um atributo e a variável, devendo ser destacado o momento em que esta associação é realizada, ou seja, o momento em que acontece o "casamento" entre a variável e cada atributo. Estes tempos podem ser: no tempo de projeto da linguagem, no tempo de implementação da linguagem, no tempo de compilação do programa, no tempo de carregamento do programa ou no tempo de execução do programa. Estes tempos são conhecidos como **tempo de vinculação** (*binding time*) e podem ter influência sobre o resultado da execução dos programas. Uma vinculação é **estática** se ocorrer antes do tempo de execução, normalmente, em tempo de compilação, e permanecer inalterada ao longo da execução do programa. Se ela ocorrer durante a execução ou puder ser alterada, é conhecida como ***dinâmica***.

O nome das variáveis.

Os nomes são utilizados para identificar variáveis, procedimentos, funções e palavras especiais da linguagem. As primeiras linguagens de programação só admitiam nomes de identificadores com apenas um caractere, a exemplo das variáveis utilizadas na Matemática. O Fortran I e Fortran 77 permitem que os nomes de identificadores tenham até 6 caracteres. O Fortran 90 e C permitem até 31 caracteres, enquanto Ada e Java permitem nomes com qualquer quantidade de caracteres e todos são significativos. C++ nada define sobre este fato. Algumas linguagens fazem diferença entre letras maiúsculas e minúsculas. Esta decisão só atrapalha a redigibilidade dos programas. É o caso de C, Java e C++, enquanto Pascal

não faz esta diferença. As palavras especiais podem ser palavras-chave se forem especiais em apenas alguns contextos, como é o caso do ***printf*** de C que só tem significado próprio se o arquivo ***stdIO.h*** estiver incluído no programa. Se uma palavra especial só puder ser utilizada para a finalidade para a qual foi definida, dizemos que a palavra é **reservada**.

O endereço das variáveis.

O endereço de uma variável é o mesmo endereço da célula na memória do computador. Uma observação sobre os endereços é que uma mesma variável pode ser associada a mais de um endereço em um programa, desde que em tempos diferentes. Por exemplo, dois subprogramas podem declarar localmente uma variável com o mesmo nome, por exemplo, *cont.* Como os subprogramas são independentes, as referências a *cont* podem acontecer sem qualquer conflito. Por outro lado, o mesmo endereço pode ser referenciado por mais de uma variável. Neste caso, dizemos que estas variáveis com o mesmo endereço são *apelidos*. Os apelidos podem trazer complicações para o programa. Por exemplo, o valor de um dos apelidos pode ser modificado sem que outro apelido tenha conhecimento disto. Normalmente, estes casos acontecem com ponteiros, mas podem ocorrer, também, em vetores ou com ***registros variantes*** de Pascal e Ada ou com as ***unions*** de C e C++.

Os valores das variáveis

O valor de uma variável é o conteúdo das células de memória associadas a ela. Por exemplo, para as variáveis inteiras são reservadas células do tamanho da palavra do computador. Em um computador em que a palavra seja de 64 bits, por exemplo, a arquitetura IA64 da Intel, para as variáveis do tipo inteiro são reservadas células de memória de 64 bits. Normalmente, estas máquinas utilizam aritmética complementar de 2 para estes valores, onde o primeiro bit, o da extrema esquerda da célula, representa o bit de sinal, sobrando 63 bits para representar os valores das variáveis inteiras, ou seja, valores entre -2^{31} a $2^{31}-1$. Para valores de outros tipos, as representações são evidentemente distintas.

Os escopos das variáveis

Antes de analisar a ideia de escopo, é necessário introduzir a ideia de blocos nas linguagens de programação. Esta metodologia foi introduzida na linguagem ALGOL 60 e permite que uma seção de código tenha suas próprias variáveis locais. Um bloco é um construtor de programa que delimita o escopo de qualquer declaração dentro dele. As variáveis declaradas em um bloco são associadas ao armazenamento no instante em que a execução do programa atinge o início do bloco, e são desfeitas, quando o bloco termina de ser executado.

Um bloco é, essencialmente, formado por uma sequência de declarações (declarações de procedimento e variáveis locais ao bloco), seguida de uma sequência de comandos, e é, normalmente, identificado pela presença de palavras reservadas ou por símbolos de início e fim de bloco, como os pares Begin e end ou { e }.

begin
    <declarações>
    <comandos>
end.

Um programa Algol é um bloco; funções e procedimentos (que correspondem às funções e sub-rotinas de FORTRAN) são blocos declarados pelo acréscimo de um *cabeçalho* a um bloco básico (begin ... end). A diferença entre uma função e um procedimento é o fato de que uma função devolve (retorna) um valor, mas, o que é dito a respeito de procedimentos se aplica também a funções.

Um *bloco anônimo*, isto é, um bloco sem cabeçalho, pode ser usado como um comando qualquer, e é entendido como um procedimento sem nome e sem parâmetros.

```
. . .
If (x > y) {
    Int z = x;
    x = y;
    y = z;
};
. . .
```

Vejamos o exemplo ao lado em Java. Neste bloco, sem nome, a variável inteira **z** é declarada, sendo visível apenas dentro deste bloco que é o seu escopo. Este tipo de bloco substitui um comando e, por este motivo, ele é conhecido como um **comando em bloco**.

É possível também uma **expressão em bloco** que é uma forma de expressão que contém uma declaração local (ou um grupo de declarações) **D** e uma subexpressão **E**. As vinculações produzidas por **D** são usadas apenas para avaliar **E**.

Nas linguagens C, C++ e Ada, o corpo de uma função é uma expressão em bloco.

Exemplo 2.1. Suponha, na figura ao lado, que as variáveis x, y e z representem as medidas dos lados de um triângulo qualquer. As expressões em bloco, em Haskell e em C++, calculam as áreas de quaisquer triângulos. A chamada em Haskell será area x y z. Em C++, será area (x, y, z);

Em Haskell:

```
area x y z =  let  s = (x + y + z) / 2.0
              in  sqrt (s * (s – x) * (s – y) * (s – z))
```

Em C++:

```
float area (float x, y, z) {
    float s = (x + y + z) / 2.0;
    return sqrt (s * (s – x) * (s – y) * (s – z));
}
```

Algumas linguagens, no entanto, só admitem blocos com um nome identificador. Este é o caso de Pascal, que só admite blocos com identificadores na forma de função (*function*) ou de um procedimento (**procedure**).

As regras de escopo

Em um programa, o escopo de uma variável é a faixa deste programa em que a variável é ***visível***. Dito de outra forma, é a faixa do programa onde a variável pode ser referenciada.

Uma variável declarada dentro de um bloco é dita ***local*** a este bloco. Se ela for declarada fora de qualquer bloco, ela é dita ***global***. As variáveis globais são visíveis em qualquer parte do programa. Pode ser que uma variável não seja local nem seja global. Isto pode acontecer se a linguagem admite escopos aninhados, como é o caso de Pascal, Módula e Ada, e for feita uma referência a uma variável cuja declaração foi feita em um bloco entre o bloco onde a referência é feita e o bloco mais externo que é o programa. Neste caso, dizemos que a variável é ***não local*** ao bloco onde ela é referenciada. A variável referenciada tem de estar em bloco a ser determinado e isso depende de que tipo de escopo é utilizado pela linguagem, ou seja, se ela usa escopo estático ou escopo dinâmico.

Escopo estático

O escopo estático recebeu este nome porque ele pode ser determinado

estaticamente, ou seja, antes da execução. Neste caso, quando uma variável é referenciada, verifica-se se existe uma declaração dela no bloco. Se existir, ela será local e tem os atributos ali declarados. Se não existir tal declaração, ela é uma variável não local e a sua declaração deve estar em um bloco na cadeia de blocos aninhados. O bloco que tem um bloco declarado dentro dele é o ***pai estático***. Pode ser que exista um avô estático, um bisavô, etc., que serão, genericamente, chamados de ***ancestrais estáticos***. O conjunto de ancestrais estáticos define a cadeia estática.

Existe também escopo dinâmico que será visto mais adiante ainda nesta seção. No entanto, verifica-se que os programas escritos em linguagens que utilizam regras de escopo estático, apresentam desempenhos melhores que os programas codificados em linguagens que utilizam regras de escopo dinâmico. No entanto, existem muitas linguagens com escopo dinâmico devido à flexibilidade que estas linguagens proporcionam. Os motivos destes fatos serão esclarecidos mais adiante neste trabalho.

Vejamos a seguir alguns exemplos que mostram a aplicação destas regras.

```
Procedure big;
        Var x : Integer;
        Procedure sub1;
            Begin    { sub1 }
             . . . . x . . . {------ ponto 1 -------}
            End;     {su1}
        Procedure sub2;
            Var x : Integer;
            Begin    { sub2 }
                 ....
            End;     { sub2  }
        Begin    {big  }
              .......
        End.     {big}
```

Exemplo 2.2. O fragmento de programa ao lado mostra um exemplo de encadeamento estático em Pascal. Neste caso, quando a variável x é referenciada, verifica-se a existência de uma declaração para ela dentro de **sub1**. Como esta declaração não existe, é feita uma busca no pai estático do procedimento onde ela é citada, ou seja, o pai estático de **sub1** que é o procedimento **big**, onde **sub1** foi declarado. Em **big** existe uma declaração para **x**; portanto, é esta a sua referência.

É preciso salientar que, em uma linguagem que utiliza regras de escopo estático, é possível que exista mais de uma declaração para a mesma variável em blocos aninhados. Neste caso, a declaração de uma variável em um bloco mais interno esconde as referências feitas a esta mesma variável,

mas que esteja em um bloco mais externo. As linguagens que adotam esta metodologia devem prover formas de acesso às variáveis ocultas mais externas. Em C e em C++, o operador de escopo "**::**" é utilizado para esta função. Se uma variável global **x** estiver oculta por uma declaração local de **x**, a variável global pode ser referenciada através da declaração **::x**. Vamos verificar, a seguir, um exemplo que mostra o processo de ocultação de variáveis em uma linguagem simples, como Pascal.

Exemplo 2.3. Vejamos o exemplo ao lado, também em Pascal, onde a variável **x**, referenciada dentro de **sub1**, no ponto 1 do programa, é local e esconde a declaração de **x** dentro do programa **main**. Se for necessário referenciar o **x** declarado em **main**, no ponto 1, deve-se usar **main.x**.

```
Program main;
    Var x : Integer;
        Procedure sub1;
            Var x : Integer;
            Begin  { sub1 }
                . . . x . . . .  {-----ponto 1 -----}
            End;   {sub1  }
        Begin { main }
            . . . . .
        End. { main }
```

Escopo dinâmico

Além do escopo estático, existe também o escopo dinâmico que é adotado por algumas linguagens. Entre elas podemos citar APL, SNOBOL, Perl, PHP e Python, além de outras. Devemos nos lembrar que, no escopo estático, um subprograma tem seu pai estático que é o subprograma onde ele foi ***declarado***. Já no escopo dinâmico, um subprograma tem seu *pai dinâmico* que é o subprograma que ***fez a chamada*** ao subprograma ou bloco, durante a execução do programa. Isto significa que no escopo estático é importante saber onde um subprograma ou bloco foi declarado e no escopo dinâmico é importante saber qual foi o subprograma ou bloco que fez a chamada ao subprograma ou bloco.

O bloco ou subprograma chamador é referenciado como ***pai dinâmico***, e, o processo de busca por uma variável não local se dá através da cadeia dinâmica, ou seja, seguindo a sequência de chamadas dos procedimentos. Neste caso, é necessário que o programa tenha registrado em algum lugar esta sequência de chamadas.

Para entender melhor esta metodologia, vamos reanalisar o mesmo exemplo mostrado anteriormente, agora à luz das regras de escopo dinâmico.

```
Procedure big;
        Var x : Integer;
        Procedure sub1;
            Begin    { sub1 }
             . . . . x . . . {------ ponto 1 -------}
            End;     {su1}
        Procedure sub2;
            Var x : Integer;
            Begin    { sub2 }
                 ....
            End;     { sub2  }
        Begin     {big  }
              .......
        End.      {big}
```

Exemplo 2.4. Seja o programa ao lado, o mesmo do *Exemplo 2.2.* Supondo que a linguagem usa agora escopo dinâmico. Vamos imaginar que a sequência de chamadas seja esta: **big** chama **sub2** que chama **sub1.**

Neste caso, a variável **x**, no ponto 1, é a declarada em **sub2**. Se a sequência de chamadas fosse **big** que chama **sub1**, a variável **x** seria a declarada em **sub1**.

A utilização de escopo dinâmico tem recebido algumas críticas dos pesquisadores. A primeira delas refere-se ao fato de que, durante a execução de um bloco ou subprograma, todas as variáveis locais a este bloco são visíveis a qualquer outro bloco ou subprograma em execução. E, não existe qualquer forma de proteger as variáveis locais dessa acessibilidade, tornando os programas dinâmicos, menos confiáveis que os estáticos.

Um segundo problema com o escopo dinâmico se refere à incapacidade de se verificar, estaticamente, os tipos das referências.

Outro problema com o escopo dinâmico é que ele torna muito difícil a leitura dos programas, porque a sequência de chamadas deve ser conhecida para se determinar o significado das referências a variáveis não locais.

Finalmente, os acessos a variáveis não locais em linguagens com escopo dinâmico levam mais tempo que em linguagens com escopo estático.

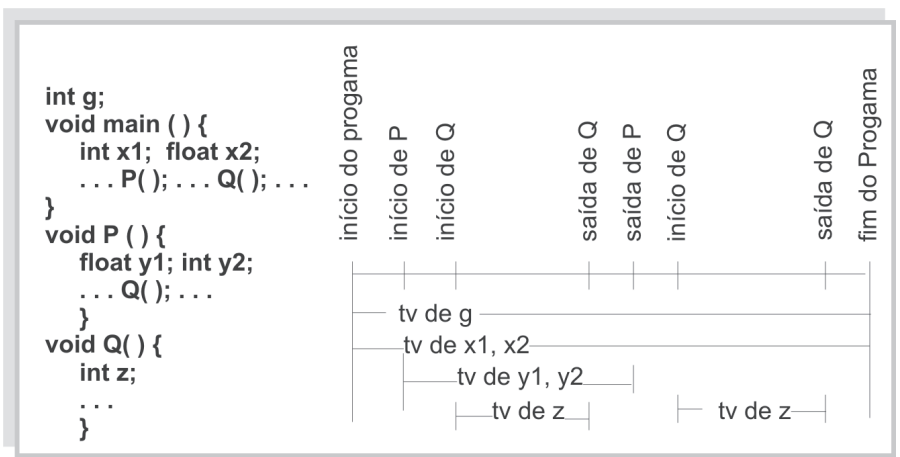
Os tempos de vida das variáveis

As variáveis são criadas ou alocadas em algum instante de tempo bem definido e são destruídas ou desalocadas em algum tempo depois, quando não forem mais necessárias. O intervalo de tempo ocorrido entre a

criação e a destruição de uma variável é conhecido como o seu tempo de vida. Uma variável só deve estar associada ao seu armazenamento, ocupando memória, durante o seu ***tempo de vida.*** Quando uma variável é destruída ou desalocada, a área de memória ocupada por ela pode ser alocada para outros propósitos. Isto significa economia de memória. As variáveis podem ser classificadas de acordo com os seus tempos de vida:

- O tempo de vida de uma variável global é o tempo em que o programa esteja em execução.
- O tempo de vida de uma variável local é o tempo de ativação do bloco em que ela foi declarada.
- O tempo de vida de uma variável heap é arbitrário, mas limitado ao tempo de execução do programa.
- O tempo de vida de uma variável persistente (uma variável arquivo) é arbitrário e pode transcender a execução de um programa.

Exemplo 2.5. Considere o programa em C ou C++; mostrado na figura a seguir. Ao lado direito é mostrado um gráfico com os tempos de vida de cada uma das variáveis declaradas no programa. No gráfico, a abreviação TV significa tempo de vida.



Classificação das variáveis

As variáveis podem ser classificadas em ***escalares*** (não estruturadas), por exemplo, byte, word, inteiras, reais e **booleanas**; e *estruturadas***,** se forem construídas a partir de outras, estruturadas ou não. As variáveis escalares podem ser classificadas em quatro tipos, de acordo com os seus tempos de vida.

Variáveis estáticas

As variáveis estáticas são vinculadas a células de memória antes de o programa iniciar sua execução, permanecendo assim, até o final da execução. Isto permite que seja utilizado o endereçamento direto, que é muito mais rápido que outros tipos de endereçamento. Além disso, pode ser necessário que uma variável local mantenha o valor anterior em uma chamada. Diz-se que esta variável deve ser sensível à história.

No entanto, este tipo de variável tem desvantagens. Por exemplo, se um subprograma tiver variáveis estáticas ele não pode ser recursivo. No Fortran I, II e IV todas as variáveis eram estáticas. Em C, C++ e em Java, a linguagem permite que o programador utilize o especificador **static** na declaração de uma variável que mantém o valor entre as chamadas.

Variáveis dinâmicas na pilha

As variáveis dinâmicas na pilha têm os seus tipos associados estaticamente, mas elas só são associadas a um armazenamento quando o bloco em que elas são declaradas iniciar a sua execução, e acaba esta vinculação quando o bloco acaba a sua execução. Este tipo de variável em um subprograma permite que ele seja recursivo. Esta denominação é devida ao fato de que a execução dos subprogramas é realizada com os registros de ativação, que são instâncias dos subprogramas, e são colocadas em uma estrutura de pilha conhecida como a pilha do sistema. Este tipo de variável permite que a área de pilha seja reutilizada para outras finalidades, permitindo compartilhamento de memória.

Variáveis dinâmicas no *heap* explícitas

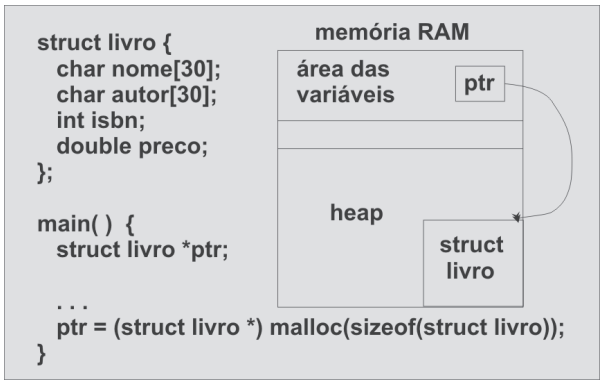
As variáveis dinâmicas no *heap* (também conhecido como monte) explícitas são células sem nomes e que se localizam em uma área da memória principal, onde o acesso a variáveis nesta área só é permitido através de variáveis do tipo ponteiro. A *heap* é uma área de memória caracterizada pela grande desorganização, porque ali as células são alocadas e desalocadas, explicitamente, mas sem qualquer ordem. Vejamos o exemplo a seguir em C, onde pode-se ver esta situação.

Exemplo 2.6. Este exemplo mostra um trecho de programa em C, mostrando o código do lado esquerdo e uma figura do lado direito representando o heap durante a execução do programa. A variável ponteiro **ptr** é alocada na área para as variáveis escalares que são dinâmicas na pilha, e o resultado da chamada a **malloc** é a alocação de uma área para a variável do tipo **struct livro**, retornando um ponteiro (endereço) para esta área que será o valor atribuído a **ptr**.

```
struct livro {
  char nome[30];
  char autor[30];
  int isbn;
  double preco;
};

main( ) {
  struct livro *ptr;

  . . .
  ptr = (struct livro *) malloc(sizeof(struct livro));
}
```



Variáveis dinâmicas no *heap* implícitas

As variáveis no *heap* implícitas são vinculadas ao armazenamento no *heap* somente quando são atribuídos valores a elas, que também é o instante da vinculação de todos os atributos da variável. Essas variáveis podem ter atributos distintos em cada valor a elas atribuído. Variáveis deste tipo têm o maior grau de flexibilidade, permitindo que se escreva um código altamente genérico. Uma grande desvantagem das linguagens que utilizam este tipo de variável é a sobrecarga necessária para manter a contabilidade destes atributos atualizada, toda vez que um novo valor é atribuído a uma variável, além de tornar impossível a detecção de erros pelo compilador. Por este motivo, poucas linguagens utilizam esta técnica. Perl, APL, JavaScript, Python e PHP são exemplos destas linguagens.

TIPOS DE DADOS

Apesar dos tipos de dados serem um dos atributos de uma variável, conforme foi visto na seção anterior, eles têm uma importância vital para a computação e, por este motivo, serão analisados de forma particular, sendo dedicada esta seção a este estudo.

Ao se declarar uma variável como sendo de um determinado tipo, na realidade se está dando duas informações importantes. A primeira delas se refere aos possíveis valores que esta variável pode assumir. A segunda informação diz respeito a quais operações podem ser aplicadas a todos os valores do tipo. Assim, existe uma correspondência entre os tipos e alguns tipos de conjuntos. A existência desta correspondência nos permite afirmar, de maneira informal, que os tipos de dados na Computação correspondem exatamente a alguns conjuntos na Matemática.

No entanto, nem todo conjunto corresponde a um tipo. É necessário que as operações entre os elementos do conjunto possam ser aplicadas a todos os elementos desse conjunto. Por exemplo, o conjunto {13, true, "José"} não representa um tipo, porque não existe qualquer operação que possa ser aplicada a todos os elementos desse conjunto. Desta forma, os tipos de dados podem ser identificados como, conjuntos matemáticos, e isto será feito neste estudo, uma vez que, a teoria matemática dos conjuntos já é bastante conhecida e desenvolvida. Esta técnica é adequada para este estudo, apesar da teoria dos conjuntos ser incompleta.

Os tipos de dados são classificados em **tipos primitivos** e **tipos compostos.**

Tipos de dados primitivos

Um tipo de dado é primitivo se seus valores não podem ser decompostos em tipos mais simples. Estes tipos podem ser os tipos numéricos (inteiros e reais), o tipo booleano e o tipo caractere.

Todas as linguagens de programação proveem tipos de dados primitivos pré-definidos, mas somente algumas delas permitem que o programador possa definir um novo tipo primitivo.

Tipos de dados primitivos pré-definidos pela linguagem

Toda linguagem de programação oferece ao programador alguns tipos pré-definidos que dão uma indicação da área de melhor e maior aplicabilidade da linguagem, sendo este um bom critério de análise de uma linguagem. Por exemplo, uma linguagem destinada a ser utilizada na área comercial, como COBOL, deve apresentar tipos primitivos cujos valores sejam strings de tamanho fixo e valores de ponto fixo. Uma linguagem destinada a ser utilizada em computação numérica, como Fortran, deve apresentar tipos cujos valores sejam números reais e tipos para valores complexos.

No entanto, alguns tipos de dados primitivos são oferecidos por muitas linguagens, mesmo com nomes distintos. Por exemplo, Java tem **boolean**, **char, int** e **float**. Ada tem os tipos **Boolean, Character, Integer** e **Float**. Os valores destes tipos são:

Bool = {false, true}
Char = { ..., ‘a’, ... ‘z’, ..., ‘0’, ... ‘9’, ..., ‘?’, ...}
Integer = { ,...,-2, -1, 0, 1, 2, ...}
Float = { ..., -1.0, ..., 0.0, ..., 1.0, ...}

Os tipos Char, Integer e Float, normalmente, são definidos por implementação, ou seja, o conjunto de valores é determinado pelo compilador. Algumas vezes estes tipos são definidos pela linguagem, ou seja, o conjunto de valores é definido pela linguagem. Por exemplo, Java define precisamente todos os seus tipos.

A ***cardinalidade*** de um tipo **T**, escrita como ***#T***, é a quantidade de valores distintos em **T**, em particular:

#Bool = 2.

#Char = 256 (Código ASCII) ou 65536 (Código Unicode).

Tipos de dados primitivos definidos pelo usuário

Para evitar problemas de portabilidade, algumas linguagens permitem que o programador defina seus próprios tipos inteiros e pontos flutuantes. Por exemplo, em Ada, o programador pode estabelecer o intervalo e a precisão de cada tipo. Por exemplo, vejamos o bloco de programa em Ada, a seguir:

```
type Populacao is range 0 . . 1e10;
pop_paiz : Populacao;
pop_mundo : Populacao;
```

Neste caso, as variáveis do tipo Populacao podem assumir valores entre 0 e 10^{10}.

Em algumas linguagens de programação, é possível definir um novo tipo primitivo inteiro, através da enumeração de seus valores. Isto é feito pelos identificadores que denotam os valores. Este tipo é conhecido como ***enumeração*** e seus valores são conhecidos como ***enumerandos***. Este é o caso da linguagem Ada.

As linguagens C e C++ também suportam enumerações, mas, nestas linguagens, as enumerações são do tipo inteiro, ou seja, cada valor representa um valor inteiro.

Por outro lado, muitas linguagens apresentam um tipo de dado ***primitivo discreto***.

Um tipo de dado é primitivo discreto é aquele cujos valores têm uma correspondência biunívoca com um subconjunto dos inteiros.

Tipos de dados compostos

Um tipo composto é aquele cujos valores são compostos a partir de outros valores mais simples. As linguagens de programação suportam uma grande variedade de tipos compostos ou estruturados como, tuplas, estruturas, registros, arrays, tipos algébricos, uniões discriminadas, objetos, uniões, strings, listas, árvores, arquivos sequenciais, arquivos diretos e relações em bancos de dados relacionais. No entanto, todos estes tipos podem ser analisados e entendidos como funções dos seguintes tipos:

- Produtos cartesianos (tuplas, estruturas e registros).
- Mapeamentos (*arrays*).
- Uniões disjuntas (tipos algébricos, uniões discriminadas e objetos).
- Tipos recursivos (listas e árvores).

O tipo produto cartesiano

O produto cartesiano é representado pelas tuplas, pelos registros e pelas estruturas de algumas linguagens.

Utilizamos a notação **(x, y)** para representar um par onde x é a primeira componente e y é a segunda. Utilizamos também a notação **S x T** para representar o produto cartesiano de S por T que é o conjunto dos pares

(x, y), sendo x escolhido em S e y escolhido em T. Formalmente, temos:

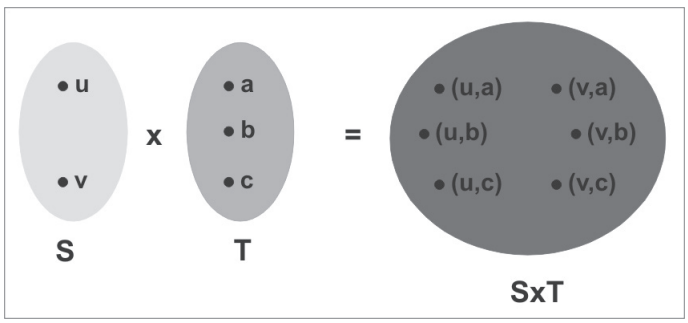
$$S \times T = \{(x, y) \mid x \in S \wedge y \in T\}$$

A cardinalidade de **S x T** é dada por:

$$\#(S \times T) = \#S \times \#T$$

A notação para o produto cartesiano de dois conjuntos pode ser estendida para representar o produto cartesiano de vários conjuntos. Neste caso, denotamos por $S_1 \times S_2 \times ... \times S_n$ cujos valores são as n-uplas onde a primeira componente está em S_1, a segunda em S_2, etc. Sendo todos os conjuntos Si do mesmo tipo, a notação anterior se torna $S^n = S \times S \times ... \times S$. Neste caso, a cardinalidade é dada por $\#(S^n) = (\#S)^n$. Considerando o caso particular em que $n = 0$, então $\#(S^0) = 1$. Isto nos informa que existe exatamente um valor para S^0. Este valor é exatamente a tupla vazia, simbolizada por **( )**, que é a única tupla sem qualquer componente.

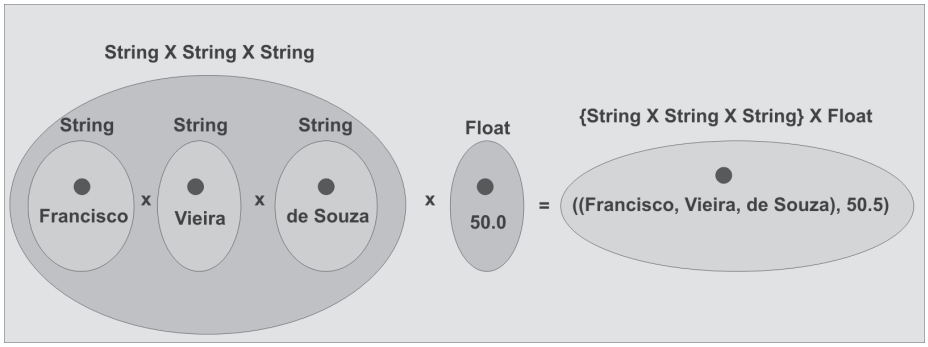
Na teoria dos conjuntos, o produto cartesiano é representado da seguinte forma:



Exemplo 2.7. Consideremos os seguintes fragmentos de programa em Cobol e em Ada:

• COBOL:	• Ada:
01 REGISTRO EMPREGADO. 02 NOME-EMPREGADO. 05 PRIMEIRO PICTURE IS X(20). 05 MEIO PICTURE IS X(10). 05 ULTIMO PICTURE IS X(20). 02 TAXA-HORARIA PICTURE IS 99V99.	type Nome_Empregado is record Primeiro : String (1..20); Meio : String (1..10); Ultimo : String (1..20); end record; type Registro_Empregado is record Nome : Nome_Empregado; Taxa_Horaria : Float; end record; Empregado : Registro_Empregado;

O tipo Empregado está mostrado na figura a seguir.



O tipo mapeamento

A noção de mapeamento de um conjunto para outro tem uma importância fundamental para a computação, uma vez que esta teoria fundamenta duas estruturas importantes nas linguagens de programação: os **arrays** e as funções.

Um mapeamento, **m**, é denotado da seguinte forma:

$$m : S \rightarrow T$$

significando que **m** mapeia todo valor de **S** em um único valor de T. Se m mapeia um valor **x** de **S** em um valor **y** de **T**, denota-se isto por **y = m(x)**, onde **y** é a imagem de **x** sob **m**. A notação **S → T** é utilizada para representar todos os mapeamentos de **S** para **T**. Formalmente, escrevemos:

$$S \rightarrow T = \{m \mid (\forall x)\ x \in S \rightarrow (\exists! y)\ y \in T \wedge y = m(x)\}.$$

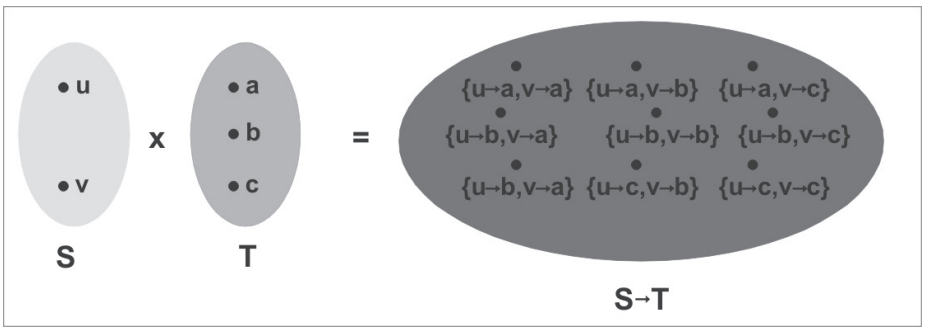
Os mapeamentos são representações de funções. Na realidade, cada par (**x, m(x)**) representa o valor de um mapeamento ou uma função total de um conjunto S sobre um conjunto **T**.

Para deduzir a cardinalidade de **S → T** é só verificar que para cada valor em S temos #T possíveis imagens. Então existem **#T x #T x ...x #T** (**#S** vezes), ou seja,

$$\#(S \rightarrow T) = (\#T)^{\#S}$$

Um ***array*** é uma sequência indexada de componentes. Ele tem um componente do tipo **T** para cada valor de **S**. O tamanho de um *array* é a quantidade de seus componentes, que é **#S**. Para os *arrays*, o conjunto **S** é finito e um subconjunto dos inteiros, que é o conjunto de índices. O menor valor e o maior valor do conjunto de índices são, respectivamente, o **limite inferior** e o **limite superior** do *array*.

Graficamente, isto pode ser visto conforme a figura a seguir.



Exemplo 2.8. Considere o seguinte fragmento em C++"

bool mat[3];

Esta declaração indica pelo visto anteriormente, que se trata de um mapeamento {0, 1, 2} → {false, true}. A cardinalidade deste ***array*** é $2^3 = 8$ valores. O elenco de todos os possíveis valores é

{{0→false, 1→false, 2→false}, {0→true, 1→false, 2→false},
{0→false, 1→false, 2→true}, {0→true, 1→false, 2→true},
{0→false, 1→true, 2→false}, {0→true, 1→true, 2→false},
{0→false, 1→true, 2→true}, {0→true, 1→true, 2→true}}

Evidente, que as funções implementadas em alguma linguagem de programação, implementam mapeamentos. Por exemplo, o código a seguir implementa um mapeamento em C++

```
bool par (int n)
  {    return (n % 2 == 0);    }
```

Implementa o mapeamento Integer → Bool com o seguinte conjunto de valores:

{ ..., 0→true, 1→false, 2→true, ...}

Vinculações dos índices e categorias de *arrays*

A vinculação dos tipos dos índices a uma variável ***array*** normalmente é estática, mas as faixas de valores dos índices, às vezes, são vinculados dinamicamente.

Em algumas linguagens de programação, o limite inferior da faixa de índices é implícito. Por exemplo, em C, C++ e em Java, o limite inferior de todas as faixas de índices é fixado em zero, enquanto em Fortran I, II, IV, Fortran 77 e Fortran 90, ele é fixado em 1. Em muitas linguagens, este valor é especificado pelo programador.

Os *arrays* ocorrem em cinco categorias. Essas categorias se baseiam na vinculação das faixas de valor dos índices e nas vinculações ao armazenamento. O nome das categorias indica onde e quando o ***array*** é alocado. São as seguintes as categorias:

- ***Array* estático:** um *array* estático é aquele no qual as faixas de índices e o armazenamento são vinculados estaticamente. A vantagem deste tipo de *array* é a eficiência, porque nenhuma alocação ou desalocação é feita em tempo de execução.
- ***Array* fixo dinâmico na pilha:** neste tipo de *array*, as faixas de índices são vinculadas estaticamente, enquanto o armazenamento é feito durante a execução. A sua vantagem sobre os *array**s*** estáticos é o compartilhamento de área de memória, porque uma matriz grande em um procedimento pode usar o mesmo espaço de memória que outra matriz, declarada em outro procedimento.
- ***Array* dinâmico na pilha:** neste tipo de *array,* as faixas de subscritos ou índices e o armazenamento são vinculados em tempo de execução. No entanto, após esta vinculação permanece fixo durante todo o tempo de vida da variável. A vantagem deste tipo de *array* sobre os anteriores está na flexibilidade, uma vez que seu tamanho não precisa ser conhecido até que esteja prestes a ser usado.
- ***Array* dinâmico no heap fixo:** este tipo de *array* é similar ao ***array*** dinâmico na pilha, no sentido de que os índices e o armazenamento são vinculados dinamicamente, mas eles são fixos após a memória ser alocada. As diferenças se referem ao fato de que as vinculações são feitas quando o programa usuário solicitá-las, em vez de serem feitas em tempo de elaboração, e a memória é alocada no heap, em vez de ser feita na pilha.
- ***Array* dinâmico no heap:** este tipo de *array* é aquele em que as vinculações dos índices e a alocação de memória são dinâmicas e podem ser trocadas durante o tempo de vida do *array*. A vantagem deste tipo de *array* sobre os outros tipos está na flexibilidade que

ele proporciona sobre os outros, uma vez que um *array* deste tipo pode crescer e diminuir durante a execução do programa.

Os *array***s** declarados nas funções em C e em C++ que incluem o modificador *static* são *array***s** estáticos. Se este modificador não estiver presente nestas funções, eles são *array***s** dinâmicos na pilha. C e C++ também possibilitam *array***s** fixos dinâmicos no heap, alocados através das funções malloc e calloc e liberadas através da função *free* em **C**, e através das funções new e delete em C++. Em Java e em **C#**, todos os *array***s** são do tipo fixo dinâmico no heap, apesar de C# também permitir a classe *Array*List que provê um *array* dinâmico no *heap*.

O tipo união disjunta

Na união disjunta um valor é escolhido a partir de diversos conjuntos possivelmente disjuntos.

A notação utilizada para a união disjunta dos conjuntos **S** e **T** é **S + T**, onde os elementos deste conjunto são compostos por um rótulo para indicar de qual conjunto o elemento é escolhido, e uma parte variante que vem ou de **S** ou de **T**. Formalmente, temos:

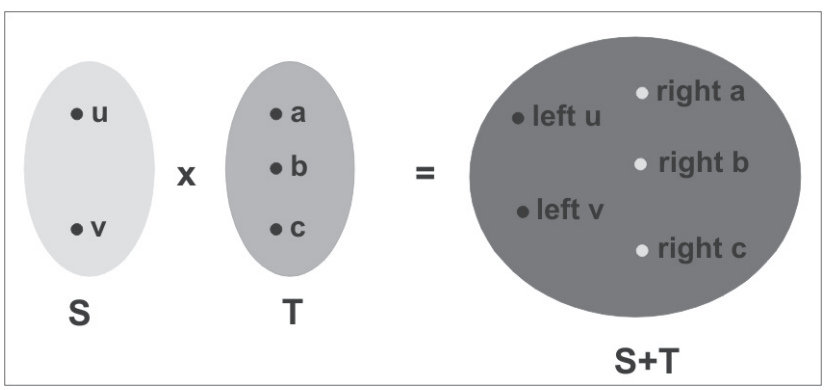
S + T = {left x | x ∈ S} U {right y | y ∈ T}

Neste caso, **left x** significa que a variante **x** vem do conjunto **S**, enquanto **right y** significa que a variante **y** vem do conjunto **T**.

A cardinalidade de **S + T** é dada por **#(S + T) = #S + #T.**

A união disjunta pode ser estendida para n conjuntos, ou seja, para $S_1 + S_2 + ... + S_n$.

A figura a seguir mostra este processo graficamente, utilizando os diagramas de Euler-Venn da teoria dos conjuntos.



Exemplo 2.9. Seja a definição de tipo a seguir em Ada graficamente, este:

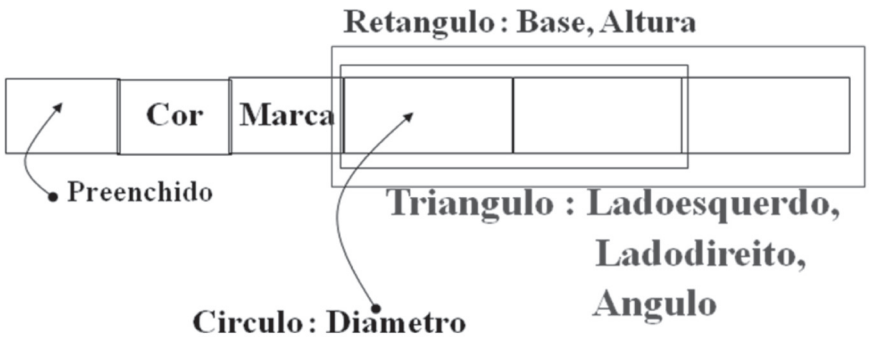
```
type Forma is (Circulo, Triangulo, Retangulo);
type Cores is (Vermelha, Verde, Azul);
type Figura (Marca : Forma) is
    record
        Preenchido : Boolean;
        Cor : cores;
        case Marca is
            when Circulo =>
                Diametro : Float;
            when Triangulo =>
                Ladoesquerdo : Integer;
                Ladodireito  : Integer;
                Angulo       : Float;
            when Retangulo =>
                Base: Integer;
                Altura: Integer;
        end case;
    end record;

Figura_1 : Figura;
```

Neste exemplo, o campo Marca é o rótulo. Dependendo de seu valor, a variável Figura_1 pode ter uma parte variante distinta. Isto significa que o tipo figura é descrito da seguinte forma:

Figura =
{Bool X Cores X Forma X Float}
+ {Bool X Cores X Forma X Integer X Integer X Float}
+ {Bool X Cores X Forma X Integer X Integer}

Possíveis valores para uma variável deste tipo podem ser:
(True, Vermelha, Circulo, 3.0), (False, Verde, Triangulo, 3, 4, 10.0), (True, Azul, Retangulo, 5, 4), além de outros valores. Uma representação gráfica para variáveis deste tipo está mostrada na figura a seguir.



Observação importante

As uniões disjuntas são tipos de dados permitidos por algumas linguagens, mas, muitas críticas são feitas a este tipo de dado. Argumenta-se que este tipo de dado introduz muita insegurança, dependendo da forma como ele é implementado na linguagem. Ada toma alguns cuidados em relação a isto, e obriga que qualquer atualização só seja permitida se for feita também ao rótulo, além de exigir que toda união tenha o rótulo. Em Pascal, o rótulo pode existir ou não. Se não existir torna este tipo em uma união livre, que é o caso da linguagem C, onde a insegurança é ainda maior, o que torna o programa muito dependente do programador, sem qualquer ajuda por parte da implementação da linguagem. Este tipo de dado se justificava em tempos onde a economia de memória era um objetivo a ser alcançado a qualquer custo. Nos dias atuais, as memórias têm se tornado bem mais baratas e este objetivo perdeu muito em importância.

Os tipos recursivos

As linguagens de programação atuais e modernas apresentam o tipo de dado recursivo, que é um dado definido em termos dele próprio. O tipo recursivo mais comum é o tipo lista, que é uma sequência de valores. Uma lista pode ter qualquer número de componentes, inclusive nenhum, que é o caso das **listas vazias**. Uma lista é um tipo de dado homogêneo, ou seja, todos os seus componentes são do mesmo tipo, apesar de cada componente poder ser de um tipo heterogêneo, como um registro, por exemplo.

Suponha que se deseja construir uma lista de valores inteiros. Neste caso, a lista ou é a lista vazia; ou é constituída de um valor inteiro, sua

cabeça, e de uma lista que pode ser vazia ou não, conhecida como a cauda da lista. Esta definição é, por natureza, recursiva:

Lista_int = nil Unit + *cons* (Integer X Lista_int)

De outra forma:

Lista_int = {nil ( )} U {cons (i, l) | i ∈ Integer, l ∈ Lista_int

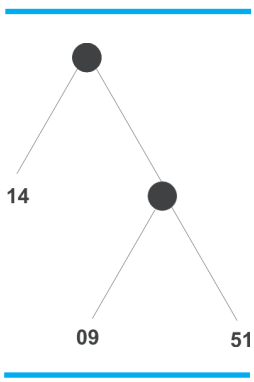
Onde ***nil*** e ***cons*** são apenas rótulos para a lista vazia e para a **lista não vazia**, respectivamente.

Normalmente, a notação ***nil* ( )** é abreviada para apenas nil. De forma generalizada, temos: **L = Unit + (T X L)** que representa o conjunto de todas as listas finitas de valores do tipo **T**. O tipo lista tem tanta importância que se dedicou a notação **T* = Unit + (T X T*)** para ele, que nada mais é do que uma nova forma de representação da mesma entidade abstrata.

Exemplo 2.10. Em Haskell, uma linguagem de programação funcional muito utilizada atualmente, uma lista de inteiros pode ser definida da seguinte forma:

data Lista_int = Nil | Cons Int Lista_int

Neste caso, a lista [3,6,9] representa Cons 3 (Cons 6 (Cons 9 Nil)). Haskell tem diversos tipos pré-definidos na linguagem, por exemplo, [Int], [String], [[Int]] que representam a lista de inteiros, a lista de Strings e a lista de listas de inteiros. Em Haskell também é possível que uma lista tenha um tipo polimórfico. Por exemplo, **[t]** representa uma lista de qualquer tipo.



Exemplo 2.11. Uma árvore binária é uma estrutura de dados que pode ser definida de forma fácil e natural em Haskell da seguinte forma:

data ArvBin t = Folha t | No (ArvBin t) (ArvBin t)

Nesta definição, cada elemento ou é um nó externo (uma Folha com um valor do tipo t) ou é um nó interno, sem um valor armazenado, mas com duas subárvores binárias como sucessoras para este nó. Por exemplo, a árvore binária ***No (Folha 14) (No (Folha 09) (Folha 51))*** pode ser representada, graficamente, como na figura ao lado.

O tipo String

Todos os tipos de dados foram vistos e analisados até o presente momento desta unidade. No entanto, cabe uma observação sobre o tipo

String.

Um String é uma sequência de caracteres, podendo ter qualquer número destes, inclusive nenhum. A quantidade de caracteres de um String é o ***tamanho*** do String. O único String que não tem qualquer caractere é o ***String vazio***.

No entanto, não existe um consenso entre os projetistas de linguagens sobre qual deve ser a forma de implementação dos Strings.

Alguns projetistas defendem que os Strings sejam implementados como tipos primitivos da linguagem. Neste caso, as operações sobre eles seriam pré-definidas e não seriam definidas na linguagem, e eles poderiam ter qualquer tamanho. A linguagem funcional ML adota esta técnica.

Uma outra metodologia que tem sido defendida por outros projetistas é considerar os Strings como *arrays* de caracteres. Neste caso, todas as operações implementadas para os *arrays*, podem ser aplicadas aos Strings. Em particular, a seleção de um caractere é um caso de indexação que é uma operação prevista para os *arrays*. Uma consequência direta desta metodologia é que todos os Strings devem ter um seus tamanhos fixados, porque os tamanhos dos *arrays* são fixados no momento em que eles são construídos. Isto proibiria que a linguagem permitisse *arrays* de tamanhos flexíveis. Esta técnica é adotada pela linguagem Ada.

Uma terceira versão destes tipos trata os Strings como *array***s** de ponteiros para caracteres, que é a técnica adotada por C e C++. Esta técnica permite uma grande economia de memória e um desempenho bem melhor no tratamento de variáveis com valores destes tipos. Deve ser esclarecido que as implementações, normalmente, são transparentes aos programadores, que não devem preocupar-se com a corretude no processamento de operações com ponteiros, por ser uma tarefa que exige uma boa experiência por parte do profissional neste tratamento.

A nosso juízo, se a linguagem de programação oferece o tipo lista, é natural implementar os Strings como listas de caracteres. Neste caso, todas as operações que são implementadas para as listas podem ser utilizadas nos Strings. Esta é a técnica utilizada por Haskell, onde uma variável String com valor “UFPI” tem o mesmo significado que a lista [‘U’, ‘F’, ‘P’, ‘I’].

EQUIVALÊNCIA DE TIPOS

Em diversas situações, devemos decidir se dois objetos são do mesmo

tipo, ou se são de tipos equivalentes, para que possam ser usados em uma determinada construção de linguagem. A forma da resposta depende, é claro, da linguagem, que deve definir suas próprias regras de equivalência de tipos.

As duas formas fundamentais de equivalência são: a equivalência nominal (dois tipos são equivalentes se têm o mesmo nome) e a equivalência estrutural (independentemente de seus nomes, dois tipos são equivalentes se têm a mesma estrutura, isto é, se são construídos da mesma maneira a partir dos mesmos tipos). Naturalmente, a primeira forma de equivalência (nominal) sempre acarreta a outra (estrutural).

Assim, os tipos das variáveis **x1** e **x2**, do **Exemplo 2.12**, a seguir, são estruturalmente equivalentes, mas não são nominalmente equivalentes; os tipos de **x1** e **y1** são equivalentes pelos dois critérios.

Exemplo 2.12. Seja o fragmento de código a seguir:

```
type
    t1 = array [1..10] of integer;
    t2 = array [1..10] of integer;
var
    x1, y1 : t1;
    x2 : t2;
```

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

Por outro lado, já observamos que a linguagem Ada adota a equivalência nominal. Ada considera de tipos distintos até mesmo duas variáveis introduzidas na mesma declaração: a declaração

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

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

Voltando ao caso bem conhecido de Pascal (cuja especificação não fixa, claramente, a forma de equivalência), temos, normalmente, implementações que misturam as duas formas de equivalência. Usando a equivalência estrutural a maior parte do tempo, e os nomes dos tipos nas especificações de tipos de parâmetros e resultado de procedimentos e funções. Em particular, inconsistentemente, valores de variáveis de tipos intervalo são tratados como pertencendo ao tipo base, sendo as restrições

verificadas apenas em tempo de execução, como seria correto se intervalos fossem apenas subtipos, como em Ada.

Diz-se que uma linguagem é fortemente tipada, se é possível, apenas por análise estática do programa (análise realizada em tempo de compilação) determinar que a estrutura de tipos da linguagem não é violada. Exemplos de linguagens fortemente tipadas são Algol-68 e Ada; Pascal seria fortemente tipada com algumas pequenas modificações, tais como a definição de tipos procedimento e a definição de "subtipos" intervalo.

Note-se que, a rigor, linguagens como Snobol e Smalltalk, cuja definição e implementação absolutamente não prevê verificação estática de tipos, podem ser consideradas (trivialmente) linguagens fortemente tipadas: basta considerar que todos os valores considerados na linguagem são do mesmo tipo (string em snobol e objeto em Smalltalk). É, assim, impossível cometer erros de tipagem, e, portanto, de forma trivial, todos os erros de tipagem podem ser detectados estaticamente. Ou seja, a propriedade de tipagem forte em linguagens de programação tem uma importância dependente da sofisticação do sistema de tipos oferecido pela linguagem.

Conversão de tipos

Se necessitarmos, em alguma situação, de um valor de um tipo t1, e apenas está disponível um valor de um tipo t2, pode ser necessária uma conversão de tipo. Vamos examinar algumas dessas situações que podem ocorrer em conjunto, em uma mesma linguagem.

- **Conversão implícita.** Neste caso, considerando-se a situação do ponto de vista abstrato, os valores do tipo t1 são valores do tipo t2, embora a representação de um valor x de t1 possa ser distinta da representação do mesmo valor x de t2. Este é o caso de inteiros e reais: abstratamente, os reais contêm os inteiros, embora as representações internas possam ser distintas. Esta é uma conversão implícita, isto é, sua necessidade não precisa ser indicada pelo programador. Por exemplo, a maioria das linguagens aceita um comando **x := i;** mesmo quando x é uma variável real, e i uma variável inteira. O código de conversão (por exemplo, do valor inteiro de i para o real equivalente) quando necessário, deve ser incluído pelo compilador. Esta forma de conversão é, às vezes, conhecida como coerção.

- **Conversão explícita.** Quando há várias maneiras de converter um valor de um tipo t1 para o tipo t2, é necessário indicar a forma de conversão desejada, e, em geral, isso é feito através de uma função de conversão. Este é o caso da conversão de reais em inteiros, que pode ser feita por arredondamento, truncamento, etc. Normalmente, funções pré-definidas da linguagem estão disponíveis para as situações mais comuns, incluindo casos como arredondamento, truncamento, aproximação superior (ceiling) ou inferior (floor).
- **Conversão formal.** Neste caso, é preciso que a conversão seja indicada, por causa do controle de tipos da linguagem, mas nenhum código precisa ser executado. Isto acontece quando temos tipos diferentes, cuja representação interna, entretanto, é a mesma. Este é o caso da função chr de Pascal, que, nas implementações usuais, não executa nenhum código, já que um caracter chr(i) é representado internamente pelo inteiro i.
- **Conversão forçada.** Por várias razões, o programador deseja que um valor do tipo t1 seja entendido como se fosse do tipo t2. Não se trata, senão formalmente, de conversão, uma vez que aqui, o que se deseja é "enganar" o compilador, o qual deve ser avisado que o valor do tipo t1 deve ser aceito como se fosse do tipo t2. Tais mecanismos se encontram em algumas linguagens, e seu uso é de exclusiva responsabilidade do programador. Em Modula-2 o nome do tipo é usado como se fosse uma função de conversão; em C, o mesmo efeito pode ser obtido através da construção sintática conhecida como typecast (molde).

EXERCÍCIO

1. Quais os principais atributos das variáveis nos programas codificados nas diversas linguagens de programação?

2. Qual o papel da vinculação nas variáveis dos programas de computador?

3. Que importância tem as regras de escopo na execução dos programas?

4. Quais tipos de dados representam as estruturas, as uniões e os vetores em C?

5. Que tipos de dados a linguagem Java apresenta?

6. Que tipos de dados as matrizes bi-dimensionais representam?

SAIBA MAIS

Existem muitos bons textos e alguns deles estão listados em Referências colocadas ao final da Unidade 2. Outros estão na Internet à disposição.

UNIDADE 3

Controle de Fluxo

Resumindo

O objetivo principal desta unidade é apresentar ao leitor os principais conceitos referentes ao controle do fluxo de um programa durante a sua execução. Este controle pode se dar em três níveis: no nível das expressões, no nível das instruções e no nível dos subprogramas. No nível das expressões, este controle é guiado pelas regras de precedência e pela associatividade dos operadores aritméticos. No nível de instrução, o controle é feito pelas instruções de se; seleção e de iteração que pode ser controlada por contadores ou por uma expressão lógica ou **boolean**a. No nível dos subprogramas, o controle é executado pelo sistema de execução que é composto pela pilha do sistema que é utilizada em cada chamada e em retornos de chamadas a subprogramas. A forma de apresentação utilizada é de acordo com o exigido para o ensino a distância, ou seja, tendo em vista sempre esta nova modalidade de ensino.

3 CONTROLE DE FLUXO

INTRODUÇÃO

A primeira forma de controle da execução de um programa acontece na avaliação das expressões. Neste caso, o controle é feito baseado nas regras de precedência e nas regras de associatividade dos operadores envolvidos na expressão.

A segunda forma de se controlar a execução de um programa se dá na execução das instruções. Neste caso, o controle se dá pela escolha da próxima instrução a ser executada. Quem comanda este controle são os comandos de seleção e de iteração que podem ser controlados por expressões lógicas ou por contadores.

A terceira forma de controle da execução de um programa acontece no nível de subprogramas. Neste caso, a pilha do sistema de execução é utilizada para proceder a chamada e o retorno de cada subprograma.

Nesta unidade, estes temas serão analisados em detalhes.

CONTROLE DE EXECUÇÃO EM NÍVEL DE EXPRESSÕES

Para entender a avaliação das expressões é necessário entender as ordens de avaliação dos operadores e dos operandos que compõem as expressões. A ordem de avaliação dos operadores é regida pelas regras de precedência e associatividade utilizadas na linguagem. Normalmente, as regras adotadas na Matemática são respeitadas, também, no processo de avaliação das expressões feitas pelos compiladores, mas isso não é uma verdade absoluta, uma vez que existem algumas linguagens que não seguem este princípio.

Avaliação de expressões aritméticas

Como já afirmado anteriormente, a maioria das características das expressões aritméticas nas linguagens de programação foi herdada de convenções desenvolvidas na Matemática. Nas linguagens de programação, as expressões aritméticas consistem de operadores, operandos, parênteses e chamadas a funções. Os operadores necessitam de operandos para que possam ser avaliados. A quantidade dos operandos que cada operador aritmético precisa para ser avaliado é conhecida como a sua **aridade**. Que pode ser **unária**, **binária** ou, mais raramente, **ternária**, conforme tenham um, dois ou três operandos, respectivamente.

Na maioria das linguagens imperativas, os operadores binários são ***infixos***, ou seja, eles são colocados entre seus operandos. No entanto, eles podem ser ***pré-fixos***, se forem colocados antes de seus operandos, ou podem ainda serem ***pós-fixos*** se forem colocados após os seus operandos. As operações pós-fixas obedecem a uma notação conhecida como ***notação polonesa reversa.*** A linguagem Perl apresenta alguns operadores pré-fixos.

Exemplo 3.1. A expressão (8 * ((5 + 4) – 10)) / 20 tem as seguintes formas de representação:

- (8 * ((5 + 4) – 10)) / 20 → Notação infixa
- / * 8 - + 5 4 10 20 → Notação pré-fixa
- 8 5 4 + 10 - * 20 / → Notação pós-fixa

Precedência de operadores

O valor de uma expressão depende, pelo menos em parte, da ordem de avaliação dos operadores na expressão. Por exemplo, em um contexto em que a = 3, b = 4 e c = 5 analisemos a expressão **a + b * c**.

Se a avaliação se proceder da esquerda para a direita, a soma será feita primeiro e depois a multiplicação. Neste caso o valor final será 35. Se, por outro lado, a avaliação se processar da direita para a esquerda, a multiplicação será realizada primeiramente e depois será feita a soma. Neste caso, a avaliação terá o valor 23. Um valor diferente do encontrado anteriormente. Mas qual é o valor correto para esta expressão? Os matemáticos decidiram que uma expressão deste tipo não deve ser avaliada, nem totalmente pela esquerda, nem totalmente pela direita. Eles criaram a ideia de precedência

dos operadores. Neste caso, a operação de multiplicação tem precedência sobre a operação de adição. Para a expressão citada, o valor correto seria 23.

As regras de precedência adotadas pela Computação, normalmente, obedecem às regras de precedência estabelecidas pela Matemática, mas esta não é uma decisão obrigatória. A tabela a seguir apresenta algumas linguagens e as regras de precedência adotadas por cada uma delas, onde a operação mais acima na tabela significa que o seu grau de precedência é maior que o de uma operação que esteja situada mais abaixo na tabela.

Fortran	**Linguagens baseadas em C**	**Ada**
**	pós-fixa ++, --	++, abs
*, /	pré-fixa ++, --, + e - unários	*, /, mod, rem
todos + e -	*, /, %	unários +, -
binários +, -		binários +, -

O operador ** é a exponenciação. O operador % de C é o mesmo operador **rem** de Ada, que toma dois operandos inteiros e retorna o resto da divisão do primeiro pelo segundo valor. O operador **mod** de Ada é idêntico a **rem** quando ambos operandos forem positivos.

Deve ser notado que a linguagem APL tem uma decisão *sui-generis* sobre este tema. Ela apresenta um único nível de precedência que é o método de avaliação da direita para esquerda.

Associatividade de operadores

A associatividade de operadores entra em cena na decisão de que operador deve ser escolhido se eles tiverem o mesmo nível de prioridade. Por exemplo, vamos analisar a expressão "**a – b + c –d"** em que as operações – e + têm os mesmos níveis de prioridade. Neste caso, deve-se fazer a subtração primeiro, ou a operação de soma? As regras de associatividade respondem a esta questão da seguinte forma. Normalmente, a associatividade utilizada pelos operadores é feita pela esquerda, exceto, quando o operador é a exponenciação. Neste caso, no exemplo em voga, a subtração a – b é realizada inicialmente, depois, soma-se esse resultado com c e finalmente diminui-se deste resultado, que será o valor final da expressão.

Alguns dados:

- O operador de exponenciação em Fortran é associativo pela direita, ou seja, na expressão A ** B ** C, avalia-se inicialmente B ** C.
- Em Ada a exponenciação não é associativa, ou seja, a expressão a ** b ** c é ambígua e não legal. Deve-se usar parênteses para resolver esta ambiguidade.

A tabela a seguir mostra as regras de associatividade adotadas por algumas linguagens de programação.

	Fortran	**Linguagens baseadas em C**	**Ada**
Esquerda	*, /, +, -	*, /, %, +, -	menos**
Direita	**	++. --, +, - unário	**

Deve ser lembrado, aqui, que as regras de precedência e associatividade podem ser alteradas pela utilização de parênteses que têm prioridade sobre todas elas.

Ordem de avaliação dos operandos

Em uma operação com mais de um operando é necessário estabelecer qual a ordem em que os operandos devem ser avaliados. Se nenhum dos operandos tiver efeitos colaterais, a ordem de avaliação não terá qualquer influência sobre o resultado final da operação. No entanto, se tiver algum efeito colateral na avaliação de algum operando, a ordem de avaliação dos operandos pode resultar em valores distintos.

Exemplo 3.2. Analisemos o programa a seguir, em C, admitindo que a função **fun** seja definida da seguinte forma:

```
int fun(int *i) {
    *i += 5;
    return 4;
}
void main( ) {
    int x = 3;
    x = x + fun(&x);
}
```

Qual o valor da variável x exatamente antes do final da execução deste programa? A resposta a esta questão pode parecer óbvia para quem não sabe que a ordem em que os operandos são avaliados tem influência no resultado final, neste caso. Se o operando a do operador + for avaliado primeiro, o resultado será 7. No entanto, pode ser que o operando fun(&x) seja avaliado inicialmente. Neste caso, a aplicação fun tem o efeito colateral de modificar o valor da variável x para 8. A soma do valor de retorno 4, com o valor de x que agora é 8, totaliza um valor igual a 12.

Conversão de tipos

Nas linguagens de programação podem acontecer dois tipos de conversão de tipos. São eles: ***estreitamento*** e ***alargamento***. Uma conversão de estreitamento transforma um valor em outro valor, de um tipo que não pode armazenar nem mesmo aproximações de todos os valores do original. Um exemplo deste tipo de conversão acontece em C, na conversão de um tipo double para um tipo float, onde a faixa de um tipo double é bem maior que a faixa de um tipo float.

Já, em uma conversão de alargamento, um valor de um tipo é convertido em um valor de outro tipo que pode incluir, pelo menos, aproximações de todos os valores do tipo original. Um exemplo deste tipo de conversão ocorre ao se converter um valor do tipo *int* em um valor do tipo *double*, em C.

As conversões podem acontecer de forma ***explícita***, feita pelo usuário, ou pode ser feita de forma ***implícita*** pelo compilador. Neste caso, o processo acontece de forma ortogonal e transparente ao programador. No entanto, é necessário que o programador conheça e domine os detalhes ocultos nos dois tipos de conversão para que seja capaz de fazer e construir programas mais confiáveis e com bom desempenho, evitando a ocorrência de sobrecargas em tempo de execução e que poderiam ser evitadas.

Operações de modo misto

Algumas linguagens de programação permitem que o programador possa codificar uma operação binária, onde um dos operandos tenha um tipo e o outro operando seja de outro tipo. Uma operação com estas características é uma **operação de modo misto**.

Nos computadores atuais, os circuitos dedicados ao processamento

de operações binárias são construídos de forma que só é possível realizar operações binárias em que os dois operandos sejam do mesmo tipo. Isso significa que os circuitos para realizar uma soma de inteiros, exigem que os valores a serem somados sejam ambos do tipo inteiro. De forma similar, também é possível somar dois números reais, e os circuitos computacionais são feitos, de tal forma, que esta operação só seja possível se ambos os valores de entrada forem números reais. Dito de outra forma, só é possível realizar operações binárias com valores do mesmo tipo.

Se for declarada uma operação binária a ser realizada com argumentos de tipos distintos, um dos valores terá que ser convertido para o tipo do outro, para que a operação possa ser realizada.

Por exemplo, se uma linguagem permitir a codificação da expressão 4.0 + 3, ou seja, a soma de um número real com um número inteiro, então, deverá acontecer uma conversão do valor 4.0 para um número inteiro para que seja somado com o valor inteiro 3; ou, o valor inteiro 3 deve ser convertido em um valor real para que possa ser somado com o valor 4.0. Este tipo de conversão é implícita, e é realizada pelo compilador de forma transparente, sem que o programador tome conhecimento deste fato. Este tipo de conversão é conhecida como **coerção**.

C++ e **Java** têm tipos inteiros menores que o tipo inteiro int. Em **C++**, eles são **char** e **short int**; em **Java**, eles são **byte, short** e **char**. Tais operandos são coagidos para o tipo **int** quando qualquer operador for aplicado a eles. Embora dados possam ser armazenados em variáveis desses tipos, eles não podem ser manipulados, antes de serem convertidos para outro maior.

Exemplo 3.3. Considere o seguinte código em Java:

```
byte x, y, z;
. . .
x = y + z;
```

Neste caso, as variáveis y e z são coagidas para o tipo int antes de ser realizada a operação de soma. Após a soma ser realizada, o valor é novamente coagido para um valor inteiro e atribuído a x.

Conversão de tipo explícita

A grande maioria das linguagens de programação permite alguma forma de conversão explícita, podendo ser de estreitamento ou de alargamento.

Nas linguagens baseadas em C, as conversões explícitas são conhecidas como **casts**. Para se especificar um *cast*, o tipo desejado deve ser colocado entre parênteses antes da expressão a ser convertida. Por exemplo, a declaração (*int*) contador; o motivo da colocação dos parênteses é que C tem vários tipos com mais de uma palavra, por exemplo, *long int*. Em Ada, a conversão explícita tem a sintaxe das chamadas de funções. Por exemplo, *Float* (soma).

Avaliação curto-circuito

Uma avaliação curto-circuito acontece em algumas linguagens, onde uma expressão não precisa avaliar todos os seus operandos para se chegar a um resultado. Por exemplo, o valor da expressão (19 * x) * (y /123 -32) independe do valor da expressão (y /123 -32) se x = 0. Porque o primeiro termo, 19 * x é 0 e 0*qualquer valor finito é 0, independente de que valor seja este. Entretanto, este atalho é difícil de ser detectado e, por este motivo, ele nunca é realizado. No entanto, este tipo de atalho é mais fácil de ser detectado em expressões booleanas. Por exemplo, seja a expressão:

(x >= 0) and (y > 20)

Se x < 0, o resultado da expressão será *false*, independente do segundo operando da expressão, porque *false and x* **=** *false*, independente do valor de ***x***.

Para ilustrar o problema potencial em avaliação de expressões booleanas, sem usar operações curto-circuito. Suponha que Java não implemente este tipo de operação e imagine uma lista de tamanho tam e codifique um fragmento de código para realizar uma busca linear nesta lista. O fragmento de código pode ser o seguinte:

Índice = 0;
while ((índice < tam) && (lista[índice] <> chave))
Índice ++;

Se não for usada a avaliação curto-circuito, os dois operandos da operação && serão avaliados. Neste caso, se a chave não estiver na lista, o primeiro argumento de && quando o último elemento da lista for analisado será avaliado como *true*, mas o valor de índice será incrementado, e, na próxima iteração, o primeiro argumento terá o valor *false*. Como a operação curto- circuito não é realizada, o segundo argumento será também analisado.

Só que o valor de índice está fora dos limites da lista, e este teste será feito sobre um valor que não está na lista.

CONTROLE DE FLUXO EM NÍVEL DE INSTRUÇÕES

As instruções de um programa são executadas em ordem sequencial, exceto, pelas instruções de desvios, que podem transferir o controle da execução de um ponto para outro na mesma unidade de programa, ou em outra. Os mecanismos de transferência entre unidades serão objeto da próxima seção.

Os mecanismos internos de transferência mais simples de serem implementados são os desvios explícitos, condicionais ou incondicionais, para pontos do programa indicados por rótulos a eles associados.

Entretanto, apesar de ser fácil compreender o efeito da execução de um destes comandos, é muito difícil visualizar o efeito de um trecho de programa onde vários destes comandos sejam utilizados, quando esses programas podem ser comparados a *"pratos de espaguete".*

Para evitar esta dificuldade, foram introduzidos os comandos *"estruturados"*, que dispensam o uso de rótulos indicativos de posições dentro de uma unidade de programa, e cujo efeito é mais fácil de ser entendido e descrito, simplesmente porque correspondem a construções habitualmente usadas.

Essencialmente, um comando estruturado se caracteriza por uma estrutura recursiva (pode ser composto de outros comandos menores). Um único ponto de entrada, um único ponto de saída, e, um efeito descrito em termos da seleção ou da repetição dos efeitos dos comandos componentes, dispensando o uso de rótulos como referências a pontos do programa. Essas estruturas recursivas têm sido também usadas em especificação informal de programas, por exemplo, em diversas formas de pseudocódigo, cuja finalidade é servir de orientação, mais ou menos detalhada, ao programador do código.

Desvios explícitos

O desvio explícito e incondicional é representado pelo comando goto, presente na maioria das linguagens, sendo ele o exemplo mais conhecido de comando não estruturado. O comando do Fortran, conhecido como "IF

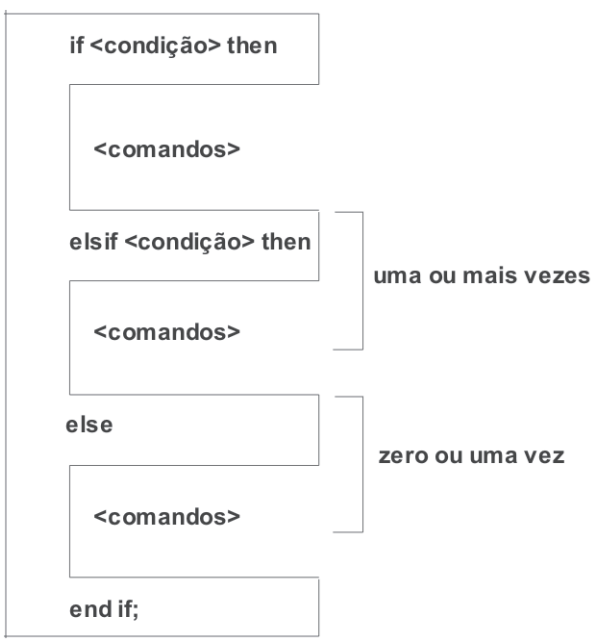
aritmético", também tem merecido atenção, mas de forma menos enfática. O IF aritmético do Fortran tem a seguinte sintaxe:

IF (<expressão>) <rótulo-1>, <rótulo-2>, <rótulo-3>

indica a transferência de controle para os comandos indicados pelos rótulos <rótulo-1>, <rótulo-2>, <rótulo-3>, conforme o valor de expressão seja, respectivamente, negativo, nulo ou positivo. As linguagens modernas apresentam formas alternativas de comando IF, que são preferíveis a estas construções. Em alguns casos, como em Fortran-77, ambas as formas coexistem na linguagem.

Comandos de seleção

A forma mais simples de comando de seleção é o comando IF de Algol ou de Pascal, que permite a execução condicional de um comando ou escolher entre duas alternativas. Linguagens como Algol-68, Modula-2 e Ada oferecem um comando "IF fechado", em que as alternativas podem ser listas de comandos, e em que mais de duas alternativas podem ser oferecidas. Este tipo de estrutura é, às vezes, chamado de *"estrutura de pente"*, sendo marcas como if**,** elsif e end if usadas para definir os *"dentes do pente"*, entre os quais se introduzem os comandos. A forma mostrada na figura a seguir é adotada em Ada:



A tabela a seguir mostra a equivalência entre as formas do comando IF de Ada e uma escadinha de IFs em Pascal. Note que, quando a lista de comandos em Ada contém mais de um comando, é necessário usar um comando composto em Pascal.

Ada	**Pascal**
if c1 then s1 elsif c2 then s2; elsif c3 then s3; s4; else s5; end if ;	if c1 then s1 else if c2 then s2 else if c3 then begin s3; s4 end else s5;

A sintaxe varia um pouco de uma linguagem para outra. No caso de Algol-68, a sintaxe do comando if se aplica também à expressões, uma vez que, comandos e expressões se confundem nessa linguagem. Por exemplo, em Algol-68, a expressão condicional: **if a=b then c else d fi;** terá como resultado c ou d, dependendo do resultado da avaliação da condição a = b**.**

Um caso especial é o dos comandos **case**, ou **switch** onde as alternativas de execução são definidas pelos valores possíveis de uma expressão. O comando **case** de Pascal também pode ser substituído por um conjunto de if's aninhados.

Usando o comando case	Usando if's aninhados
case exp of v1 : c1; v2 : c2; v3 : c3; ... end;	v:=exp; if v=v1 then c1 else if v=v2 then c2 else if v=v3 then c3 ...

No exemplo acima, exp é uma expressão; v1, v2, v3 são os valores esperados de exp; c1, c2, c3, ... são as alternativas de comandos correspondentes. A variável temporária v é usada apenas para evitar que a expressão seja calculada mais de uma vez.

Outras linguagens acrescentam uma alternativa (else em Modula-2, otherwise em PL/I, others em Ada, default no comando switch em C), para o caso em que exp não tenha qualquer dos valores v1, v2, v3, ... Além disso, normalmente, são permitidos intervalos de valores como opções. No caso do comando switch da linguagem C, devemos observar que a semântica envolvida é diferente, como pode se ver no exemplo a seguir.

Exemplo 3.4. Seja a declaração a seguir de uma seleção *switch* em C:

```
switch ( exp ) {
case v1: s1;
case v2: s2;
case v3: s3; break;
case v4: s4; break;
default: s5;
}
```

Neste exemplo, verifica-se a presença do comando **break** nas alternativas v3 e v4. A sua finalidade, neste caso, é encerrar a execução do comando **switch**. Como no caso de Pascal, a expressão exp só é avaliada uma vez, mas, diferentemente da metodologia ali adotada, mais de uma alternativa pode ser escolhida em uma mesma execução do comando. O

uso do comando break é necessário para finalizar o switch nesta linguagem. Suponha que exp tenha o valor v2, e que os quatro valores v1, v2, v3 e v4 sejam todos distintos. Neste caso, a execução se realiza da seguinte forma:

- a expressão exp é avaliada, obtendo-se o valor v2;
- compara-se o valor de exp com v1; como são diferentes, passa-se ao teste seguinte (até encontrar um valor igual ou o rótulo default);
- compara-se o valor de exp com v2; como são iguais, s2 é executado;
- a execução continua no comando seguinte, s3;
- ao chegar ao comando break a execução do comando switch é interrompida, continuando a execução do programa a partir da instrução seguinte a ele.

Para C, portanto, a lista de comandos dentro do comando switch é apenas uma lista de comandos rotulados, que devem ser executados em sequência, exceto, por dois pontos:

- o primeiro comando a ser executado é aquele rotulado pelo valor da expressão, ou por default, se o valor da expressão não corresponder a nenhuma alternativa;
- o comando break, se estiver presente, encerra a execução do comando switch inteiro. Isto não é um privilégio do comando switch; o comando break pode também ser utilizado para interromper a execução de outros comandos da linguagem.

O comando **case** pode ser implementado de forma semelhante ao aninhamento de if's, ou seja, por comparações sucessivas com os diversos rótulos, ou através de uma tabela de desvios para os comandos correspondentes, indexada pelo valor da expressão. A execução é mais rápida neste segundo caso, porque depois de avaliar a expressão, basta desviar para o endereço correspondente na tabela.

A principal vantagem deste comando é o aumento de legibilidade, que fica muito prejudicada quando é necessário considerar um grande número de alternativas usando comandos if.

Execução iterativa de instruções

As instruções de iteração consistem em fazer com que uma instrução ou um conjunto delas, seja executada, zero, uma ou mais de uma vez. Todas as linguagens imperativas incluem um ou mais tipos de iteração. Se este tipo de instrução não existisse, o programador teria que declarar cada ação em sequência e os programas seriam muito grandes e demorariam muito tempo para serem codificados.

Uma **construção de iteração** é composta de uma **instrução de iteração** e de um **corpo** do laço que é a coleção de instruções controladas pela instrução de iteração.

Nas linguagens imperativas, a execução do corpo do loop pode ser controlada por uma expressão booleana, por um contador ou por uma combinação destes dois métodos.

Nas linguagens funcionais, como Haskell, as execuções iterativas são controladas através da definição de funções recursivas.

Nas instruções iterativas onde o controle seja feito por uma expressão lógica, o controle da iteração pode ser feito antes ou depois da execução do corpo do laço. Se for realizado antes, o mecanismo de controle é conhecido como pré-teste e se for feito após a execução do corpo do laço, será chamado de pós-teste.

Nos comandos de repetição, desejamos a execução repetida, de forma controlada, de uma ação ou série de ações. A ação indicada é executada, ou enquanto se mantiver verdadeira uma condição de continuação, ou enquanto uma condição de término se mantiver falsa. Nesta seção, serão discutidos os seguintes comandos de repetição: while, repeat, loop e for.

Loops controlados por expressões lógicas

Um comando while exp do c; precede cada execução do comando c pela avaliação e teste do valor da expressão exp; c só é executado se o valor de exp for true. Portanto, exp especifica uma condição de continuação da repetição. Como no caso do if, algumas linguagens usam uma construção fechada, em que c pode ser uma lista de comandos: while exp do c end.

Um comando semelhante é o comando do-while que é definido por do c while exp; que difere do comando anterior pelo fato de que c é sempre executado uma primeira vez. Por seu lado, no comando repeat c until exp, a

expressão exp é uma condição de término: a repetição continua até que exp se torne verdadeira. Também neste caso, a lista de comandos c é sempre executada pelo menos uma vez.

Até certo ponto, é possível intercambiar os usos dos comandos repeat e while, pela negação da expressão e. Assim, um comando repeat c until exp é equivalente à sequência de comandos

```
c;
while not exp do
    c
```

e o comando while exp do c é equivalente à combinação

```
if exp then
    repeat
        c
    until not exp
```

Loops controlados por contadores

Por outro lado, as instruções de iterativas podem ser controladas por um ou mais contadores. Por exemplo, o comando

for v := e1 to e2 do c

tem sua execução controlada pelos valores da variável v, conhecida como a *variável de controle* do for. Inicialmente, os valores v1 e v2 de e1 e e2 são obtidos. Se v1<v2, c é executado uma vez para cada valor v=v1, v=v2; caso contrário, c não é executado nenhuma vez. O comando **for** é equivalente a:

```
inf := e1;
sup := e2;
v1 := inf;
while v <= sup do begin
c;
v := v + 1;
end;
```

onde inf e sup são variáveis temporárias, que garantem que as expressões e1 e e2 sejam avaliadas apenas uma vez.

Com esta definição (usada em Pascal) vemos que, se n > 0,

for v := 1 to n do c

especifica a execução de c exatamente n vezes. No caso do comando do de Fortran, isto não ocorre: a lista de comandos é sempre executada pelo

menos uma vez. Em Algol-60, podemos definir, através da cláusula step s, que o passo, isto é, o incremento da variável do for a cada interação deve ser s. Em Pascal, existem apenas duas opções de passo: +1, indicado pelo uso de to, como acima, e -1, indicada pelo uso da palavra downto, caso em que a variável de controle assume, sucessivamente, valores decrescentes.

O comando For visto acima, se distingue fundamentalmente dos comandos while e repeat, pelo fato de que no For as expressões são pré-calculadas, e o número de execuções é conhecido, a priori, enquanto, no caso dos comandos while e Repeat, as expressões são reavaliadas a cada vez. Linguagens como Ada e Clu consideram que a variável v do For é declarada pelo cabeçalho, e é, portanto, local ao comando. Nessas linguagens não há nenhuma indeterminação quanto ao valor da variável v após a execução do comando: esse valor não existe. Outras linguagens costumam indicar que "a variável do FOR tem um valor indefinido após a execução do comando", para que possa ser escolhida, em cada caso, a melhor forma de implementação.

Em Algol-60, por exemplo, encontramos várias outras formas de comando For. Em particular, é possível especificar uma condição de continuação, através de uma cláusula **while** e3, em vez de especificar um valor final para a variável: a execução do comando se encerra quando a avaliação de e3 levar a um valor falso

for v:= e1 step e2 while e3 do c

As duas formas podem ser combinadas:

for v:= e1 step e2 while e3 until e4 do c

A linguagem C apresenta um comando For que separa claramente os seus quatro componentes:

for (e1; e2; e3) s

onde:

- e1 especifica a forma de iniciação;
- e2 especifica a condição de continuação (a execução continua enquanto a avaliação de e2 obtiver um valor interpretado como true);
- e3 prepara a execução da próxima iteração, por exemplo incrementando o valor de uma variável de controle;
- s é o comando a ser repetidamente executado.

Por exemplo, para executar o comando s para os valores 1,2, ... , n da variável i, teríamos:

for (i=1; i<=n; i++) s.

Considerado extremamente poderoso pelos programadores de C, este comando é bem característico do estilo compacto de programação da linguagem C, e, juntamente com outros comandos da linguagem, tem sido criticado por levar a programas obscuros, dependendo de sua forma de utilização.

Os comandos Loop e Exit

Em algumas situações, os comandos de repetição já apresentados são inadequados para refletir a estrutura desejada. Por essa razão, algumas linguagens (Ada, Modula-2, por exemplo) introduzem um comando de repetição indefinida **loop**, cuja execução só deve ser interrompida por um comando exit, que transfere o controle da execução para o ponto imediatamente após o loop, terminando assim sua execução. No caso geral, a combinação loop-exit não tem equivalentes simples em termos de while e repeat. A combinação loop-exit satisfaz os requisitos mencionados para um comando estruturado, uma vez que tem uma única saída, mesmo quando vários comandos exit forem utilizados. A implementação dos comandos loop e exit é idêntica à do comando goto, ficando a diferença por conta das restrições em seu uso.

Cada comando de repetição controlado logicamente, pode ser simulado por outros comandos, também, controlados logicamente. Os comandos de repetição controlados logicamente são mais completos que os comandos de repetição controlados por contadores. Isto significa que qualquer comando de repetição controlado por contadores pode ser simulado por um comando controlado logicamente, mas a recíproca não é verdadeira. O exemplo a seguir mostra como um comando de repetição controlado logicamente é simulado por outros.

Exemplo 3.5. A tabela a seguir apresenta uma implementação de um loop e três simulações deste mesmo loop utilizando instruções de iteração do tipo repeat, while e goto.

loop *c1;* *if e then* *exit;* *c2;e* *end;*	*repeat* *c1;* *fim := e;* *if not fim then* *c2;untilL fim;*
c1; *while not e do* *begin* *c2;*	*ini:* *c1;* *if e then* *goto fim;*
c1; *end;*	*c2;* *goto ini;* *fim:*

Em Ada, a forma mais geral exit r when e; especifica a condição e para que o comando loop de rótulo r tenha sua execução interrompida. A forma mais simples do comando é simplesmente exit; e especifica a interrupção do loop mais interno, de forma incondicional. Embora não ofereça um comando com as propriedades dos comandos loop de Ada ou Modula-2, C oferece o comando break, já visto anteriormente, e que também permite a saída imediata de um ponto qualquer dentro de um dos diversos comandos de repetição.

CONTROLE DO FLUXO EM NÍVEL DE SUBPROGRAMAS

Este é o terceiro nível em que acontece o controle da execução dos programas. Os subprogramas representam a abstração de processos, que é um dos dois tipos de abstração em que as linguagens de programação se baseiam. O outro tipo de abstração se relaciona aos dados, de onde surgiu a programação orientada a objetos, um tema a ser tratado na próxima unidade.

Um subprograma consiste em um conjunto de instruções que são tratadas como uma unidade única e pode ser chamado em várias partes de um programa, economizando tempo de programação e tornando os programas menores e mais fáceis de serem codificados. Os subprogramas podem ser funções ou procedimentos. Uma função representa uma abstração de uma função matemática e, como tal, a sua chamada sempre produz um resultado. Já os procedimentos representam a abstração de um comando e

a sua execução não produz um resultado, na grande maioria das linguagens de programação.

Fundamentação dos subprogramas

Os subprogramas que serão tratados neste estudo têm algumas características comuns a todos. São elas:

- Cada subprograma tem um único ponto de entrada.
- Na execução de um subprograma, a unidade chamadora é suspensa, ficando ativo somente o subprograma chamado.
- Após a execução de um subprograma, o controle sempre retorna para a unidade chamadora, exatamente no ponto seguinte ao ponto em que foi feita a suspensão.

Algumas linguagens, por exemplo, Fortran, podem ter múltiplas entradas, no entanto, esta decisão de seus projetistas não representa diferenças de capacidade e de análise.

Definições iniciais

Um subprograma é constituído de alguns componentes que devem ser definidos, para que possam ser reconhecidos nas diversas linguagens. São eles:

- **Definição.** A definição de um subprograma envolve uma interface para comunicação entre o programa e o subprograma, sendo esta comunicação feita através de parâmetros de entrada e de retorno, que são variáveis utilizadas apenas para este fim.
- **Chamada.** Uma chamada a um subprograma é uma solicitação explícita para que o subprograma seja executado. Diz-se que um subprograma está ativo se ele foi chamado, foi iniciada sua execução, mas ainda não foi concluída.
- **Perfil.** O perfil de parâmetros em um subprograma se relaciona com o número, a ordem e os tipos dos parâmetros formais.
- **Protocolo.** O protocolo de um subprograma se refere ao perfil mais o tipo do retorno, se for uma função. Pode existir, ou não.
- **Cabeçalho**. O cabeçalho de um subprograma é constituído de uma indicação sobre o tipo do subprograma, normalmente,

indicado por uma palavra especial da linguagem, do nome do subprograma e do *protocolo*, se existir.

Os parâmetros dos subprogramas

Existem duas formas com as quais os subprogramas se comunicam com o programa principal ou com outros subprogramas: por acesso direto a variáveis não locais declaradas em outros subprogramas, mas que sejam visíveis no local, ou pela passagem de parâmetros. O acesso a variáveis não locais de forma descontrolada pode comprometer a confiabilidade do sistema, porque pode acontecer que alguns acessos não sejam desejáveis.

Os parâmetros no cabeçalho de um subprograma são chamados de ***parâmetros formais*** também conhecidos como ***variáveis fictícias***.

As chamadas aos subprogramas incluem o nome do subprograma e uma lista de valores conhecidos como ***parâmetros reais*** que são vinculados aos ***parâmetros formais***.

Na grande maioria das linguagens, nas chamadas a subprogramas deve existir uma correspondência biunívoca entre os parâmetros formais e os parâmetros reais, no que se refere à ordem em que os parâmetros reais são colocados. Neste caso, o primeiro parâmetro real deve corresponder ao primeiro parâmetro formal e assim por diante. Neste caso, estes parâmetros são ditos ***posicionais***. Se a lista de parâmetros for muito grande é fácil se cometer um erro na ordem dos parâmetros.

Para evitar este tipo de erro, algumas linguagens permitem os parâmetros de ***palavra-chave***, onde cada parâmetro formal se vincula a um parâmetro real. Este tipo de parâmetro permite que a ordem dos parâmetros reais seja alterada. Por exemplo, a linguagem Ada permite a seguinte declaração

```
Somador (comprimento => meu_comprimento,
                vetor => meu_vetor,
                soma => minha_soma);
```

As linguagens C++, Fortran 90 e Ada permitem que os parâmetros formais tenham valores padrão. Um valor padrão é usado se nenhum parâmetro real for passado ao parâmetro formal na chamada ao subprograma. Por exemplo, o cabeçalho a seguir, em Ada:

```
function calc_pagamento (renda : float;
                         isenções : integer := 1;
                         tarifa_imposto : float) : return float;
```

O parâmetro isenções pode estar ausente em uma chamada a calc_pagamento. Neste caso, o valor 1 é utilizado para ele.

A linguagem C++ não permite a declaração de parâmetros de palavra-chave, mas permite valores padrão, só que com regras distintas, onde os parâmetros padrão devem aparecer por último, porque os parâmetros são associados posicionalmente. Uma vez que um parâmetro padrão seja omitido em uma chamada, todos os parâmetros formais restantes têm valores padrão.

Mecanismos de passagem de parâmetros

Diversos mecanismos de passagem de *parâmetros* tem sido definidos e implementados, de acordo com critérios de projeto ou de implementação de cada linguagem. Vamos fazer aqui uma exposição geral desses mecanismos, e discutir finalidades, forma de implementação, vantagens e desvantagens de cada um.

Chamaremos, aqui, de parâmetros, as variáveis utilizadas na declaração de unidades de programa com a finalidade específica de fazer referência a objetos externos; os argumentos, associados aos parâmetros durante a chamada. Evitamos assim, os nomes parâmetro formal e parâmetro real (ou “atual”) que, além de não concordarem com o uso normal em matemática, se prestam à confusão. Observamos que um argumento só faz sentido no ambiente de execução do procedimento chamador (a unidade de programa que efetua a chamada), da mesma forma que só faz sentido o parâmetro no ambiente de execução do procedimento chamado. A passagem de parâmetros visa exatamente permitir a referência a valores da unidade chamadora, no ambiente da unidade chamada.

Passagem por referência

Este é o mecanismo mais simples de passagem de parâmetros: o procedimento chamado tem acesso, através dos parâmetros, aos próprios argumentos. Para isso, o parâmetro recebe como valor o endereço do argumento, e, o procedimento chamado pode ler ou alterar o valor do

argumento, através de seu endereço. Em linguagens com outras formas alternativas de passagem de parâmetros, pode-se restringir a passagem de expressões por referência. Por exemplo, em Pascal, não podem ser passadas por referência expressões que não são variáveis, e essa restrição é enfatizada pelo uso da palavra reservada var, que indica a passagem por referência apenas de variáveis.

Casos em que não existe uma forma alternativa de passagem de parâmetros aceitam-se como argumentos passados por referência, expressões que não são variáveis, e, em particular, valores constantes. Como não faz sentido alterar o valor de tais argumentos, a maioria das implementações protege o argumento contra a alteração de valor, através de sua duplicação: o parâmetro recebe apenas o endereço de uma variável temporária, cujo valor inicial é aquele obtido pela avaliação da expressão (efetivamente uma cópia do valor, no caso de uma constante) e pode ser alterado sem problemas. Em particular, a alteração inadvertida do valor de uma constante traria efeitos imprevisíveis para o restante da execução do programa.

Passagem por valor

No caso da passagem por valor, apenas o valor do argumento se torna disponível, e o parâmetro é, de fato, uma variável local ao procedimento, que, por ocasião da chamada, recebe, como valor inicial, o valor do argumento já calculado no ambiente do chamador. Nenhuma ação é tomada para a devolução de qualquer valor ao chamador. O parâmetro é uma variável local, e não há, em princípio, nenhuma proibição da alteração de seu valor.

Normalmente, a passagem por valor não deve ser a única disponível, uma vez que não prevê a volta de resultados para a unidade chamadora. Em algumas linguagens (C, Algol-68), é possível passar por valor uma referência (endereço) do argumento. Neste caso, mesmo que o endereço se mantenha inalterado, o conteúdo do endereço pode ser alterado. Em outras linguagens, a passagem por valor aparece combinada com a passagem por resultado, que veremos a seguir. O Exemplo 3.6 mostra como em C a passagem por valor do endereço de uma variável permite a emulação de uma passagem por referência.

Exemplo 3.6. Seja o seguinte programa em C.

```
void incr(int *x; int delta)     --  x é referência a inteiro
{
*x += delta;                     --  conteúdo de x é incrementado
}
int y=15;
int cinco=5;
...                              --   aqui, y = 15
incr( &y, cinco );               --  argumento é o endereço de y
...                              --  aqui, y = 20
```

Passagem por resultado

De certa forma simétrica à anterior, a passagem por resultado caracteriza o parâmetro como uma variável local, cujo valor final, no instante em que o procedimento se prepara para retornar o controle da execução ao chamador, é passado para o argumento. O argumento, neste caso, deve ser uma variável. Este mecanismo é semelhante àquele utilizado por uma função para devolver seu resultado; a diferença consiste na notação, e o parâmetro pode não ter nome: é uma variável temporária com a finalidade de receber este valor, e permitir sua transmissão para a unidade chamadora. Em algumas linguagens, o próprio nome da unidade é usado para esse parâmetro, em outras, indica-se o valor a ser devolvido através de uma palavra chave, tal como ***result*** ou **return**.

Passagem por valor-resultado

Neste caso, a passagem por valor aparece combinada com a passagem por resultado: o parâmetro é entendido como uma variável local inicializada com o valor do argumento, e cujo valor final é passado ao argumento, no término da execução. Como na passagem por referência, é possível consultar e alterar o valor de uma variável, mas, as duas formas não são equivalentes, como veremos na comparação entre formas de passagem de parâmetros, mais abaixo.

Passagem por nome

Esta forma de passagem considera um aspecto ainda não examinado: não só o valor de um argumento pode se alterar durante a execução da unidade chamada, mas o próprio argumento pode se modificar. Por exemplo, se o argumento é a componente de array y[i+1,j], e as variáveis i e j têm seus valores alterados durante a execução, a variável cujo nome é y[i+1,j] também se modifica, e pode, em princípio, ser qualquer uma das componentes do array y.

Na passagem por nome, o programador entende o parâmetro como sendo o argumento que tem aquele nome, em cada instante da execução, e não o argumento que tinha aquele nome por ocasião da chamada. Assim, o parâmetro reflete as alterações de valor do argumento, e as alterações do próprio argumento, quando seu nome (fixo) passa a descrever outra variável.

Para o programador, tudo se passa como se o código (fonte) do procedimento chamado fosse copiado, substituindo-se cada uso de um parâmetro pelo nome do argumento correspondente. Naturalmente, essa regra da cópia não pode, em geral, ser diretamente aplicada, sem levar em conta a diferença dos ambientes de execução: o mesmo nome pode significar coisas distintas em distintos ambientes, e o cálculo do valor do argumento só pode ser feito no ambiente do chamador.

A unidade chamadora não pode fornecer, no momento da chamada, o valor do argumento ou seu endereço, uma vez que isto não seria adequado. De acordo com a implementação mais típica, a unidade chamadora fornece a forma de calcular os argumentos correspondentes aos nomes dados, através de trechos de código construídos com essa finalidade. Esses trechos (*thunks*) podem ser vistos como procedimentos sem parâmetros, que calculam o valor ou o endereço de um argumento, conforme apropriado; cada *thunk* é ativado quando necessário pela unidade chamada, mas é sempre executado no ambiente de execução da unidade chamadora.

O Exemplo 3.7 mostra como a passagem por nome pode ser utilizada, de forma difícil de simular nas outras formas de passagem de parâmetros. A um programador não acostumado com a passagem por nome, dificilmente ocorreria a ideia de dar a todas as componentes de um vetor v o valor zero, chamando um procedimento como o procedimento zero, cujo único argumento é uma componente genérica v[i] do vetor v.

Nenhuma das outras formas de passagem de parâmetros permitiria o

acesso a uma componente genérica de v[i]: por valor, seria passado o valor de v[i], para o valor inicial de i; por referência, seria passado o endereço de v[i], também para o valor inicial de i. No caso da passagem por nome, o que é efetivamente passado é a forma de cálculo do endereço da i-ésima componente de v, v[i].

Exemplo 3.7. Seja o seguinte fragmento de programa em Pascal.

```
var
    i:integer;
    v:array[1..100] of integer;
    procedure zera(j:integer);
            begin
        for i:=1 to 100 do
        j:=0;
           end;
   ...
zera(v[i]);
```

Comparação. As diversas formas de passagem de parâmetros correspondem, como vimos, a diferentes formas de implementação, e conduzem a diferentes resultados, que, entretanto, na maioria dos casos, não oferecem surpresas. Por exemplo, podemos distinguir entre passagem por referência e passagem por valor/resultado com o procedimento incr2 do Exemplo 3.8.

Exemplo 3.8. Seja o seguinte fragmento de programa em Pascal.

```
procedure incr2(x,y:integer);
begin
    x:=x+1;
    y:=y+1;
end;
...
z:=0;
incr2(z,z);
...
```

Se for usada a passagem de parâmetro por referência, o valor de z

após a chamada será 2; se for usada a passagem de parâmetros por valor/resultado, o valor será 1.

A passagem por referência foi uma das primeiras metodologias de passagem de parâmetros a ser utilizada, e é a forma oferecida por Fortran. Em Algol-60, a passagem por nome é default mas é também oferecida a passagem por valor, para acelerar a execução dos programas em que uma longa sequência de chamadas (por exemplo, de um procedimento recursivo) implicaria num número grande de chamadas de *thunks*, com prejuízo da velocidade. Pascal, em virtude da preocupação com simplicidade que presidiu ao seu projeto, oferece as duas maneiras mais simples de passagem de parâmetros: por valor (default) e por referência (indicada por var). O caso da linguagem Ada é interessante: os parâmetros podem ser declarados de três modos: in, out, in out, que aparentemente correspondem à passagem por valor, resultado, e valor-resultado. O manual da linguagem, entretanto, abre a possibilidade de que o modo in out seja implementado por referência, e adverte o programador que um programa cujo funcionamento correto dependa da forma de implementação é um programa Ada errôneo. Segundo o mesmo manual, um programa errôneo é um programa incorreto, mas nenhum compilador de Ada precisa indicar este fato. Por exemplo, um programa Ada semelhante ao do Exemplo 3.9 seria errôneo. O Exemplo 21 mostra como simular passagem por nome através de outras formas.

Exemplo 3.9. Suponhamos que se deseja calcular $\sum xi$ para $m \leq i \leq n$, através de uma função soma, que deve receber como parâmetros i, m, n, xi: soma(i, m, n, xi). Com a passagem por nome, podemos escrever:

```
integer function soma(i, m, n, x);
   integer i,m,n,x;
   begin
      integer s;
      s:=0;
      for i:=m to n do s:=s+x;
      soma:=s;
end;
```

Por exemplo, para calcular $\sum\sum A[i, j]$, $1 \leq i \leq m$, $1 \leq j \leq n$ deveríamos chamar soma (i,1,m,soma(j,1,n,A[i,j])).

Se quisermos implementar o mesmo problema em Pascal, teremos

que definir uma função que calcule o valor de xi, e passe essa função como argumento para a função soma.

```
function soma(var i, m, n : integer; function x : integer) : integer;
var s:integer;
begin
    s:=0;
    for i:=m to n do s:=s+x;
    soma:=s;
end;
```

Neste caso, para calcular o duplo somatório, teríamos que definir uma função Aij, que calcularia o valor da componente i,j de A.

soma(i,1,m,soma(j,1,n,Aij)

A função Aij deve apenas atrasar o cálculo de A[i,j], esperando que i e j tenham atingido os valores apropriados:

```
function Aij:integer;
begin
    Aij:=A[i,j]
end;
```

Note que não há necessidade de alterar a forma de passagem dos demais parâmetros de soma (i, j, m), porque, tratando-se de variáveis simples, os efeitos da passagem desses parâmetros por nome e por referência se equivalem.

Subprogramas sobrecarregados

Um subprograma é sobrecarregado se ele tiver várias implementações. Isto significa que o mesmo subprograma atua sobre tipos de dados distintos, implementando algoritmos também distintos. Por exemplo, o operador de multiplicação * em C é utilizado para multiplicar dois números reais e também para multiplicar dois números inteiros. A multiplicação de inteiros é bem diferente da multiplicação de números reais, porque as representações em bits de números reais são diferentes da representação de números inteiros, e os circuitos dos computadores dedicados a estas operações são distintos.

Alguns projetistas de linguagens argumentam que a sobrecarga não

aumenta o poder de expressividade de uma linguagem, exatamente porque implementa algoritmos distintos. Se eles são distintos, podem ter nomes também distintos.

No entanto, muitos projetistas argumentam que alguma forma de sobrecarga deve existir pelo menos para operações que tenham alguma coisa em comum, como a multiplicação de números, e muitas linguagens adotam esta técnica.

Por exemplo, C++, Java e Ada incluem subprogramas sobrecarregados pré-definidos e os programadores também têm permissão para escrever múltiplas versões de subprogramas com o mesmo nome nessas linguagens.

Subprogramas genéricos

A reutilização de software tem representado uma técnica de programação importante, notadamente, na implementação de grandes sistemas, em que a programação é feita por grupos ou times de profissionais. A Engenharia de Software tem se preocupado com o gerenciamento destes grupos e times, empregando várias técnicas para isto.

Neste caso, a reutilização pode ser incrementada pela construção de subprogramas que possam ser utilizados em mais de um tipo de dados, mas, fazendo exatamente a mesma implementação. ***Subprogramas genéricos*** ou ***polimórficos*** aumentam a produtividade do software pelo fato de poderem ser aplicados a vários tipos de dados. Isto significa que o mesmo subprograma pode ser aplicado a vários tipos de dados. Os subprogramas sobrecarregados apresentam uma espécie particular de polimorfismo chamado de ***polimorfismo ad hoc.***

Corrotinas

Uma corrotina é um tipo especial de subprograma em que não existe a reação mestre-escravo entre o chamador e o chamado. O modelo de controle de corrotinas, frequentemente é chamado de modelo de controle simétrico. Este modelo de controle foi implementado pela primeira vez em Simula 67.

Ao contrário dos subprogramas comuns, as corrotinas têm vários pontos de entrada que são controlados pelas próprias corrotinas, e têm meios de manter sua história entre as ativações. As execuções das corrotinas

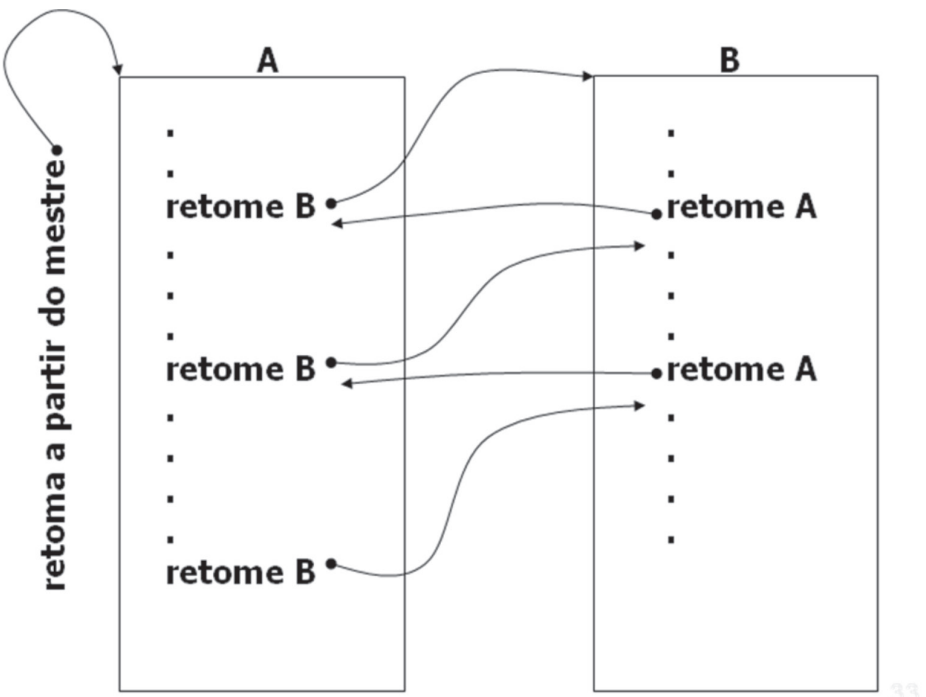
normalmente têm início em pontos distintos do início, sendo este, o motivo pelo qual se chama de retomada da rotina, em vez de chamada.

Tipicamente, as corrotinas são criadas em uma aplicação por uma unidade de programa chamada de ***unidade mestra***, que não é uma corrotina.

Um exemplo clássico que simula a execução de corrotinas é um jogo de cartas em que participam quatro jogadores, utilizando a mesma estratégia de jogo. O programa mestre inicia a simulação, retomando uma das corrotinas jogadoras, que, depois de haver jogado sua vez, poderia retomar a seguinte e assim por diante até o jogo terminar.

A Figura 3.1 a seguir, mostra graficamente uma possível sequência de execução de duas corrotinas A e B. Na figura pode-se observar os vários pontos de entrada e de saída que as corrotinas têm. A execução de programas em tempo compartilhado, onde, para cada processo é destinado um time slice, dá-se através de corrotinas. Uma vez que, cada processo tem de ficar suspenso por algum período e tem de retornar ao ponto em que foi suspenso para continuar a sua execução, quando for dado a ele um novo *time slice*.

Figura 3.1. Esquema de execução de duas corrotinas.



EXERCÍCIO

1. Que são valor-l e valor-r das variáveis?

2. Que é um apelido no contexto das linguagens de programação?

3. Que são as variáveis estáticas, no contexto das linguagens de programação?

4. Que são as variáveis dinâmicas na pilha, no contexto das linguagens de programação?

5. Que são as variáveis dinâmicas no heap explícitas, no contexto das linguagens de programação?

6. Que são as variáveis dinâmicas no heap implícitas, no contexto das linguagens de programação?

7. Que é coerção de tipos?

8. Quais os tipos de iteração utilizados pelas Linguagens de Programação? Qual é a mais genérica?

9. Qual a diferença entre o comando for de Java e de C++?

10. Quais os principais métodos de passagem de parâmetros utilizados pelas linguagens de programação em seus subprogramas?

11. Qual a diferença entre um procedimento e uma função nas linguagens de programação?

12. O que são as corrotinas e quais as suas diferenças com relação aos subprogramas?

13. Qual a diferença entre sobrecarga e polimorfismo?

SAIBA MAIS

Existem muitos bons textos sobre este tema. Alguns deles estão listados em Referências e colocados ao final desta Unidade. A nosso ver, um livro importante é o livro de David Watt, onde ele realça a ligação entre a Matemática e a Ciência da Computação em termos da teoria dos conjuntos e dos tipos de dados. Outros estão na Internet à disposição. Estes estão listados a seguir.

UNIDADE 4

Abstração de Dados

Resumindo

O objetivo principal desta unidade é apresentar os principais conceitos envolvidos nas abstrações de dados e como eles são utilizados nas linguagens de programação. Inicialmente, será vista uma análise dos tipos abstratos de dados e como eles podem ser utilizados para resolver o problema da crise do software dos anos 80. Em um segundo momento, a análise se estende às linguagens de programação projetadas para incorporar as características dos tipos abstratos de dados, conhecidas como linguagens orientadas a objetos. Os objetos serão vistos como tipos abstratos de dados que incorporam mais uma característica que é herança. A unidade termina com uma análise das principais linguagens de programação orientadas a objetos. A forma de apresentação utilizada é de acordo com o exigido para o ensino a distância, ou seja, tendo em vista sempre esta nova modalidade de ensino

4 ABSTRAÇÃO DE DADOS

INTRODUÇÃO

O termo Programação Orientada a Objetos foi criado por Alan Kay, autor da linguagem de programação Smalltalk. Mas mesmo antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar estas ideias foi a linguagem Simula 67, criada por Ole Johan Dahl e Kristen Nygaard em 1967. Este paradigma de programação já é bastante antigo, mas apenas ultimamente vem sendo aceito, realmente, nas grandes empresas de desenvolvimento de software. Alguns exemplos de linguagens modernas, utilizadas por grandes empresas em todo o mundo, que adotaram essas ideias: Java, C#, C++, Object Pascal (Delphi), Ruby, Python e Lisp.

A maioria delas adota as ideias, parcialmente, dando espaço para o antigo modelo procedural de programação, como acontece no C++, por exemplo, onde temos a possibilidade de usar POO, mas a linguagem não força o programador a adotar este paradigma de programação, sendo ainda possível programar da forma procedural tradicional. Este tipo de linguagem segue a ideia de utilizar uma linguagem previamente existente como base e adicionar novas funcionalidades a ela.

Outras são mais “puras”, sendo construídas do zero, focando-se sempre nas ideias por trás da orientação a objetos, como é o caso das linguagens Smalltalk, Eiffel, Self e IO, onde TUDO é orientado a objetos.

O CONCEITO DE ABSTRAÇÃO

Uma abstração significa uma visualização ou a representação de uma entidade, incluindo apenas os atributos que sejam realmente importantes

em um contexto particular. Isto permite que se construam grupos destas entidades, que guardem entre si, atributos comuns. Estes atributos comuns podem ser abstraídos, e, dentro de cada grupo, somente os atributos que distinguem os elementos individuais precisam ser considerados. Isto permite que os elementos do grupo sejam simplificados. A abstração é uma arma contra a complexidade na programação, por simplificar e muito este processo.

Consideram-se dois tipos de abstração nas linguagens atuais: a abstração de processos e a abstração de dados.

A abstração de processos

Este conceito é mais antigo que o de dados e mesmo as linguagens mais antigas o incorporam. Os subprogramas são abstrações de processos, porque permitem que um processo seja executado sem a necessidade de especificá-lo em cada ponto de chamada. Por exemplo, quando se precisa ordenar um vetor em uma execução de um programa, constrói-se um subprograma para realizar esta tarefa, imaginando-se que outros vetores também necessitem ser ordenados futuramente no programa. Neste caso, apenas o subprograma é chamado tendo um novo vetor como parâmetro, não sendo necessário, novamente, escrever todo o código para fazer esta ordenação. A abstração de processos permite que programas grandes possam ser lidos e entendidos mais facilmente. A principal motivação para a abstração de dados é a mesma da abstração de processos. Busca resolver o problema da complexidade dos grandes programas.

O conceito de abstração de dados constitui o objetivo principal desta unidade e será visto em todo o decorrer deste estudo. Para entender de forma plena as suas funcionalidades é necessário, antes, conhecer alguns outros conceitos importantes que são pré-requisitos para o domínio desta metodologia de programação.

O CONCEITO DE ENCAPSULAMENTO

Já foi mencionada em outras unidades a existência da programação simples, conhecida como programming in the small e a programação mais complexa, denotada como programming in the large. Nesta última, quando os programas representam algumas centenas de milhares de linhas de código, surgem alguns problemas de ordem prática.

O primeiro deles se refere à dificuldade que o programador terá na administração destes sistemas de forma coerente e confiável, sendo necessária uma organização em módulos, que são grupos de subprogramas e dados que estejam logicamente relacionados.

O segundo problema diz respeito à recompilação, após alguma modificação em alguma parte do programa. Nos programas pequenos este problema é praticamente inexistente, mas, em um sistema grande, o custo de recompilação se torna significativo. É necessário encontrar formas de evitar que as unidades que não foram modificadas sejam também recompiladas. Isto pode ser conseguido organizando-se os programas em coleções de subprogramas e de dados, em que cada uma destas coleções possam ser compiladas, sem a necessidade de que as outras também sejam. Esta coleção é conhecida como unidade de compilação.

O ***encapsulamento*** é um agrupamento de subprogramas e dos dados que eles manipulam. Ele constitui um sistema abstraído e uma organização lógica para uma coleção de computações relacionadas. Ele resolve os dois problemas elencados anteriormente.

Os encapsulamentos são, muitas vezes, incorporados a bibliotecas e colocados à disposição para serem reutilizados em programas que não foram projetados para utilizar estas bibliotecas.

Em muitas linguagens de programação do tipo Algol, os programas podem ser aninhados, como, por exemplo, Pascal. Tal método de organização de programas utilizando regas de escopo estático tem sido muito condenado, exatamente pelo fato de que os subprogramas não podem ser recompilados separadamente. Ou seja, os subprogramas aninhados não criam boas construções de encapsulamentos.

Na linguagem C, uma coleção de funções e definições de dados relacionadas pode ser colocada em um arquivo, que pode ser compilado de forma independente. No entanto, as definições de dados em arquivos diferentes não são verificadas. Isto significa que os arquivos em C não criam encapsulamentos seguros.

Fortran 90 e Ada permitem que coleções de subprogramas de tipos e de dados sejam agrupadas em unidades que podem ser compiladas, separadamente, de forma que as informações de interface sejam salvas pelo compilador e usadas na verificação de tipos, quando usadas por outra unidade. Essas unidades fazem encapsulamentos perfeitos.

TIPOS DE DADOS ABSTRATOS DEFINIDOS PELO USUÁRIO

Formalmente, um tipo de dado abstrato definido pelo usuário é um tipo de dado que satisfaz as duas seguintes condições:

- A representação ou a definição do tipo e as operações sobre instâncias do tipo estão contidas em uma única unidade sintática. Além disso, outras unidades de programa têm permissão para criar variáveis do tipo definido.
- A representação de uma instância do tipo não é visível pelas unidades de programa que usam o tipo, de modo que as únicas operações diretas possíveis sobre essas instâncias são aquelas oferecidas na definição do tipo.

As unidades de programa que usam um tipo de dado abstrato específico são chamadas clientes desse tipo.

As vantagens de empacotar a representação e as operações em uma mesma unidade sintática são as mesmas já vistas para o encapsulamento. A principal delas é que os clientes não são capazes de ver os detalhes da representação e seu código não pode depender dessa representação. Em consequência disso, as representações podem ser modificadas a qualquer instante sem exigir modificações nos clientes. Além do mais, o fato dos clientes não poderem modificar as representações, resulta em programas mais confiáveis, aumentando a integridade das instâncias desses tipos de dados.

Para construir um tipo abstrato de dado é necessário, antes de tudo, definir que operações o tipo de dado vai oferecer aos clientes deste tipo. Por exemplo, para definir um tipo abstrato pilha é necessário definir as operações mostradas na tabela a seguir. Estas operações são as mais comuns em uma estrutura deste tipo. Os nomes estão em inglês por ser o uso comum e a tradução pode gerar dúvidas por parte dos clientes.

Tabela 4 1. Principais operações de uma pilha.

create(stack)	Cria e possivelmente inicializa uma pilha
destroy(stack)	Desaloca o armazenamento da pilha
empty(stack)	Uma função que verifica se a pilha está vazia ou não
push(stack, element)	Coloca um elemento na pilha
pop(stack)	Remove o elemento do topo da pilha
top(stack)	Retorna uma cópia do elemento do topo da pilha

Tipos de dados abstratos em Ada

As construções de encapsulamento em Ada são chamadas de pacotes, que são compostos de duas partes: a especificação do pacote onde é feita definição da interface do encapsulamento e o corpo do pacote, onde são feitas as implementações das entidades nomeadas na especificação. Não é obrigatória a presença do corpo do pacote nos tipos abstratos de Ada, se ele não for necessário. Por exemplo, pacotes que encapsulam apenas dados e constantes não precisam ter um corpo.

A especificação de um pacote e o seu corpo têm o mesmo nome. O corpo de um pacote é identificado pela palavra reservado **package** co cabeçalho do corpo. A especificação e o corpo de um pacote podem ser compilados separadamente, desde que a especificação seja compilada antes do corpo.

Exemplo 4.1. Vamos mostrar uma definição de uma pilha como um tipo abstrato de dado em Ada. As operações para o tipo pilha são as mostradas na Tabela 4.1.

```
package Stack_Pack is -- as entidades visíveis aos clientes
type Stack_Type is limited private;
      Max_Size : constant := 100;
function Empty (Stk : in Stack_Type) return Boolean;
procedure Push(Stk : in out Stack_Type;
                     Element : in Integer);
procedure Pop (Stk : in out Stack_Type);
function Top (Stk : in Stack_Type) return Integer;
private
   type List_Type is array (1 .. Max_Size) of Integer;
   type Stack_Type is
```

```
        record
            List : List_Type;
            Topsub : Integer range 0 . . Max_Size := 0;
            end record;
    end Stack_Pack;
```

Deve ser observado que nenhuma operação de criação ou de destruição foi incluída porque elas não são necessárias. O corpo do pacote Stack_Pack é definido da seguinte forma:

```
    with Ada.Text_IO; use Ada.TExt_IO;
    package package Stack_Pack is
      function Empty(Stk : in Stack_Type) return Boolean is
        begin
          return Stk.Topsum = 0;
      end Empty;
     procedure Push(Stk : in out Stack_Type;
      Element : in Integer) is
      begin
        If Stk.Topsum >= Max_Size
            then Put_Line("ERROR – Stack overflow");
            else
                Stk.Topsum := Stk_Topsub + 1;
                Stk_List(Topsub) := Element;
       end if;
     end Push;

     procedure Pop(Stk : in out Stack_Type) is
       begin
         If Stk.Topsub = 0
            then Put_Line("ERROR – Stack underflow");
            else Stk.Topsub := Stl.Topsub – 1;
         end if;
       end Pop;

     function Top(Stk : in Stack_Type) return Integer is
       begin
         If Stk.Topsub = 0
             then Put_line("ERROR – Stack is empty");
```

```
            else return Stk.List(Stk.Topsub);
        end if;
    end Top;
end Stack_pack;
```

O procedimento Use_Stacks, a seguir, é um cliente do pacote Stack_pack. Ele ilustra como o pacote pode ser usado.

```
with Stack_Pack, Ada.Text_IO;
use Stack_Pack, Ada.Text_IO;
procedure Use_Stacks is
  Topone : Integer;
   Stack : Stack_Type; cria uma instância de Stack_Type
   begin
     Push(Stack, 19);
     Push(Stack, 30);
     Topone := Top(Stack);
     Pop(Stack);
     . . .
  end Use_Stacks;
```

Tipos abstratos de dados em C++

A linguagem C++ foi criada pela adição de suporte à programação orientada a objetos. Ada oferece encapsulamento para simular tipos de dados abstratos, através dos pacotes, enquanto C++ oferece as classes que são tipos. Em C++ as variáveis são declaradas como instâncias de uma classe, tornando-as mais parecidas com os tipos.

As classes em C++ podem conter entidades ocultas, declaradas pela cláusula private, e também entidades públicas declaradas pela cláusula public.

Em C++ dispõe-se de uma função constructor na definição de uma classe, usada para inicializar dados dinâmicos no heap e também de uma função destructor, chamada, implicitamente, para encerrar o tempo de vida de uma instância de uma classe, usada com o sinal ~.

Exemplo 4.2. O mesmo tipo abstrato pilha, implementado em Ada no Exemplo 4.1, será aqui mostrada sua implementação em C++, para que o leitor possa fazer uma comparação entre as duas implementações.

```
#include <iostream.h>
class stack {
  private:
      int *stackPtr;
      int max_Size;
      int topPtr;
   public:
     stack( )  {
        stackPtr = new int [100];
        max_Size = 99;
        topPtr = -1;
      }
       ~stack( )  {delete [ ] stackPtr;};   //** função destrutora
    void push (int numbar) {
        If (topPtr = = Max_len)
           cerr << “Error in push-stack is full\n”;
       else stackPtr[++topPtr] = number;
 }
void pop( ) {
    If (topPtr = = -1)
      cerr << Error in pop-stack is empty\n”;
      else topPtr- -;
}
int top( ) {return (stackPtr[topPtr]);}
int emptry( ) {return (topPtr = = -1);}
}
```

Para utilizar este tipo de dado em C++, o programa a seguir mostra uma possibilidade.

```
void main( ) {
   int topOne;
   stack stk;
   stk.push (19);
   stk.push (30);
   topOne = stk.top( );
   stk.pop(.):
```

```
    . . .
}
```

Tipos abstratos de dados em Java

O suporte da linguagem Java para os tipos abstratos de dados é similar ao de C++, com apenas algumas diferenças. Todos os tipos de dados abstratos definidos pelo usuário em Java são classes, já que Java não inclui structs, e todas as instâncias são alocadas no heap e são acessadas via variáveis do tipo referência. Outra diferença é que os métodos de Java só podem ser definidos em classes. Assim, um tipo de dado abstrato em Java é declarado e definido em uma mesma unidade sintática.

Em vez de ter as cláusulas privadas e públicas nas definições de classes, em Java, os acessos podem ser incorporados aos métodos e nas definições de variáveis.

Exemplo 4.3. O mesmo tipo de dado pilha, já visto anteriormente, implementado em Ada e em C++, será aqui mostrada sua implementação em Java.

```
import Java.io.*;
class StackClass {
    private int [ ] stackRef;
    private int maxLen, topIndex;
    public StackClass( ) {    // Um construtor
       stackRef = new int [100[;
       maxLen = 99;
       topIndex = -1;
    }
    public void push (int number) {
       If (topIndex = = maxLen)
          System.out.println(“Error in pop-stack is empty”);
       else - - topIndex;
    }
    public int top( ) {return (stackRef[topIndex]);}
    public boolean empty( ) {return (topIndex = = -1);}
}
```

O fragmento de código a seguir, em Java, mostra uma forma de utilização deste tipo abstrato de dado.

```
public class TstStack {
   public static void main(String[ ] args) {
      StackClass myStack = new StackClass( );
      myStack.push (19);
      myStack.push (30);
      System.out.println("30 is:  "   + myStack.top( ));
      myStack.pop ( );
      System.out.println("19 is:  "   + myStack.top( ));
      myStack.pop ( );
      myStack.pop ( );    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETOS

A programação orientada a objetos tem sua origem em Simula 67, mas ficou estabelecida a partir da linguagem Smalltalk, considerada por alguns a única linguagem verdadeiramente orientada a objetos.

A programação orientada a dados concentra-se nos tipos de dados abstratos. Neste caso, as operações a serem realizadas em um dado são feitas através de funções ou métodos associados a este dado. Por exemplo, se um vetor precisa ser classificado, ele é declarado como um tipo de dado abstrato, onde a classificação do vetor é feita através de uma função de classificação associada ao tipo de dado. As linguagens que suportam a programação orientada a dados e incorporam um mecanismo de herança e um tipo particular de vinculação dinâmica, ficaram conhecidas como linguagens orientadas a objetos.

Em POO, as classes são declarações de objetos, que poderiam também ser definidas como abstrações de objetos. Isto quer dizer que a definição de um objeto também é a definição da classe da qual os objetos são instâncias desta classe. Quando definimos um objeto com suas características e funcionalidades, em verdade, o que estamos definindo é uma classe. Portanto, para um cliente, as classes são tipos abstratos de dados que herdam.

Desta forma, antes de investigarmos outras construções das linguagens orientadas a objetos, vamos analisar os suportes que fundamentam a programação orientada a objetos. Vamos iniciar pelo mecanismo de herança.

HERANÇA

Com o advento da programming in the large, os desenvolvedores de software verificaram ser importante desenvolver sistemas que pudessem ser reutilizados em soluções de problemas semelhantes. Os tipos abstratos de dados foram evidentemente os principais candidatos a realizar este objetivo, devido ao encapsulamento e ao controle de acesso que eles permitiam. No entanto, para ser utilizado como solução para um novo problema, um tipo de dado abstrato teria que ter alguma modificação e isto exigiria que o programador que fosse fazê-la, tivesse conhecimento de parte ou de todo o código existente. Além disso, essas modificações exigiriam mudanças em todos os programas clientes.

Um segundo problema com as definições em tipos abstratos de dados é que elas são todas independentes e estão no mesmo nível. Isto, frequentemente, torna impossível estruturar um programa que se ajuste ao espaço de um outro problema. Em muitos casos, os problemas exigem a construção de estruturas com certo grau de hierarquia.

O mecanismo de herança oferece uma solução tanto para o problema de modificação apresentado pela reutilização de tipos abstratos, quanto para o problema de organização do programa. Se um novo tipo de dado puder herdar os dados e a funcionalidade de um tipo existente e modificar algumas entidades existentes e acrescentar algumas próprias, a reutilização será aumentada grandemente, sem exigir modificações no tipo de dado abstrato reutilizado.

Vamos imaginar um sistema para a Biologia, modelando o homem como um animal. O homem tem todas as características (atributos) dos animais e pode realizar todas as ações (métodos) de um animal. Além disso, o homem também tem algumas outras características e ações inerentes apenas a ele.

Para simular este contexto, vamos criar a classe Animal e a classe SerHumano como uma subclasse da classe Animal. A subclasse SerHumano tem todas as características e comportamentos da classe Animal e mais algumas outras adicionais inerentes apenas ao ser humano. Neste caso, a subclasse SerHumano herda o estado e o comportamento da classe Animal.

A classe SerHumano será uma especialização da classe Animal. A classe Animal é a classe pai da subclasse SerHumano, e logicamente, a classe SerHumano é a classe filha da classe Animal.

Uma classe pode sempre ter vários filhos. Uma classe pode ter apenas um pai, como Smalltalk, o que caracteriza herança simples, no entanto, a linguagem C++ permite que uma classe herde as características de várias classes, caracterizando a herança múltipla. Deve ser lembrado, no entanto, que C++ não é um bom exemplo quando se está falando sobre conceitos de POO. C++ é, na realidade, a linguagem C com uma extensão para orientação a objetos, o que é diferente de uma linguagem que foi projetada para ser orientada a objetos, como Smalltalk.

Como outro exemplo um pouco mais próximo da nossa realidade, vamos imaginar um sistema para um banco comercial. O banco possui clientes que são pessoas físicas e jurídicas.

Poderíamos criar uma classe chamada Pessoa com os atributos Nome e Idade. Em seguida, criamos duas classes que serão filhas da classe Pessoa, chamadas PessoaFisica e PessoaJuridica. Tanto a classe PessoaFisica como Pessoa Juridica herdam os atributos da classe Pessoa, mas também, podem ter seus próprios atributos a mais. Por exemplo, a classe PessoaFisica pode ter o atributo RG, enquanto a classe PessoaJuridica pode ter o atributo CNPJ. Assim, os objetos da classe PessoaFisica terão como atributos: Nome, Idade e RG, enquanto os objetos da classe PessoaJuridica terão os atributos: Nome, Idade e CNPJ.

Com relação aos métodos, pode-se agir de forma semelhante. Poderíamos criar alguns métodos na classe Pessoa e mais alguns nas classes PessoaJuridica e PessoaFisica. No final, todos os objetos teriam os métodos especificados na classe Pessoa, mas só os objetos do tipo PessoaJuridica teriam os métodos especificados dentro da classe PessoaJuridica, e objetos do tipo PessoaFisica teriam os métodos especificados na classe PessoaFisica.

Estados em objetos

Quando temos um objeto suas propriedades tomam valores. Por exemplo, quando temos um carro a propriedade cor tomará um valor concreto, como, por exemplo, preto, cinza, etc. O valor concreto de uma propriedade de um objeto chama-se estado.

Para acessar o estado de um objeto, ver o seu valor ou atualizá-lo, em muitas linguagens, utiliza-se o operador ponto. Por exemplo,

meuCarro.cor = amarela;

que, deve ser interpretado como, “a cor do objeto meuCarro será amarela”.

Neste caso, estamos atualizando o valor do estado da propriedade do objeto para amarela, com uma simples atribuição.

Mensagens em objetos

Uma mensagem em um objeto é a ação de efetuar uma chamada a um método. Por exemplo, quando dizemos ao objeto carro para andar, estamos lhe passando a mensagem “ande”.

A forma utilizada pelas linguagens para enviar mensagens aos objetos é a mesma utilizada para atualizar os estados dos objetos. Na maioria das linguagens, é utilizada a notação de ponto. Por exemplo, o comando

```
meuCarro.andar();
```

significa que estamos passando a mensagem andar() para o objeto meuCarro. Deve-se colocar parênteses, como em qualquer chamada a uma função, colocando dentro deles os parâmetros.

Exemplo 4.4. Seja o caso de uma locadora de vídeos com diversos clientes. Poderíamos então criar uma classe Cliente com as seguintes características (atributos): Nome, Data de Nascimento e Profissão.

Um cliente é mais que simples dados. Ele pode realizar ações! E no mundo da POO, ações são descritas através de métodos.

Dessa forma, objetos da classe Cliente poderão, por exemplo, executar as seguintes ações (métodos): AlugarFilme, DevolverFilme e ReservarFilme.

Note o tempo verbal empregado ao descrever os métodos. Fica fácil perceber que tratam-se de ações que um Cliente pode realizar.

É muito importante perceber as diferenças entre Atributo e Método. No começo, é normal ficar um pouco confuso, mas tenha sempre em mente que “atributos são dados” e “métodos descrevem ações que os objetos são capazes de realizar”.

Assim, nosso sistema pode ter vários Objetos do tipo Cliente. Cada um destes objetos possuirá seu próprio nome, data de nascimento e profissão, e todos eles poderão realizar as mesmas ações (AlugarFilme, DevolverFilme ou ReservarFilme).

De forma resumida, os objetos têm estados, representados por seus campos, e também tem comportamentos, representados por seus métodos.

POLIMORFISMO

Um dos conceitos mais complicados de se entender, e também um dos mais importantes, é o **Polimorfismo**. O termo é originário do grego e significa “muitas formas”.

Na orientação a objetos, polimorfismo significa que um mesmo objeto, sob certas condições, pode realizar ações diferentes ao receber uma mesma mensagem. Isto significa que apenas olhando o código fonte não é possível saber exatamente qual será a ação a ser tomada pelo sistema, sendo que, o próprio sistema é quem decide qual método será executado, dependendo do contexto durante a execução do programa. Desta forma, a mensagem “fale” enviada a um objeto da classe Animal, pode ser interpretada de diferentes formas, dependendo do objeto em questão. Para que isto ocorra, é preciso que duas condições sejam satisfeitas: exista herança de uma **classe abstrata** e **Casting** (outras situações também podem resultar em polimorfismo, mas vamos nos centrar neste caso). Estes termos serão explicados a seguir.

CLASSES ABSTRATAS

Uma classe abstrata é uma classe que representa uma coleção de características presentes em vários tipos de objetos, mas que não existe e não pode existir isoladamente. Por exemplo, podemos criar uma classe abstrata chamada **Animal**. Um Animal tem diversas características (atributos) e pode realizar diversas ações (métodos), mas não existe a possibilidade de criarmos objetos do tipo Animal. O que existem são objetos das classes Cachorro, Gato, Papagaio, etc. Essas classes estendem a classe Animal, herdando todas as suas características, e adicionando algumas coisas a mais. **“Animal”** é só uma **entidade abstrata**, apenas um conjunto de características em comum, nada mais.

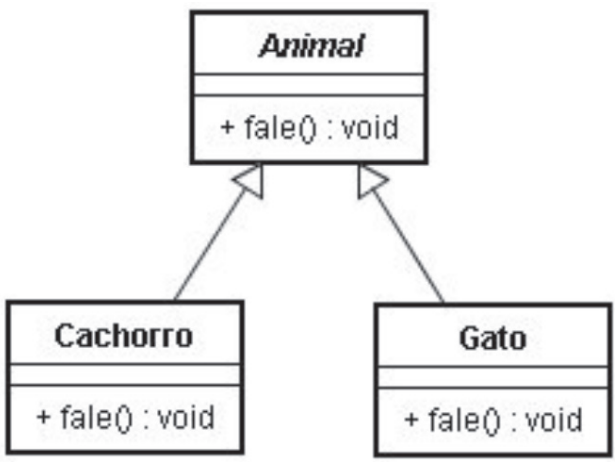
Você pode olhar para um objeto da classe Cachorro e falar “isto é um animal, pois estende a classe Animal”, mas você nunca vai ver um objeto que seja apenas da classe Animal, pois isso não existe! É como eu olhar para você e falar “você é um objeto da classe SerVivo”. Essa afirmação está correta, mas você na verdade é um objeto da classe Ser Humano, que, por sua vez, herda todas as características da classe SerVivo (que, por sua vez, é uma classe abstrata, já que não podemos criar algo que seja apenas classificado como “SerVivo”, sempre vamos classificá-lo de forma menos genérica). Resumindo: uma classe abstrata é um conjunto de informações a

respeito de uma coleção de outras classes. Uma classe abstrata sozinha é completamente inútil, já que não podemos instanciar um objeto desta classe, podemos apenas instanciar objetos de classes que estendem a classe abstrata inicial. Ela serve apenas para simplificar o sistema, juntando em um único lugar diversas características que são comuns a um grupo de classes.

Não esqueça disso: você nunca vai poder criar um objeto do tipo de uma classe abstrata. Sempre crie objetos das classes que estendem a classe abstrata.

Casting em POO

O termo **Casting** é utilizado quando nós forçamos o sistema a ver um objeto como sendo de um determinado tipo, que não é o seu tipo original. Suponhamos a situação mostrada na figura a seguir:



Essa é uma representação na notação UML (*Unified Modeling Language*), que nos informa que existe a definição de três classes em nosso sistema: existe a classe Animal (note que ela está em itálico; isso significa na notação UML, que se trata de uma classe abstrata) e existem, também, outras duas classes chamadas **Cachorro e Gato**, que são **filhas da classe Animal**. Todo objeto das classes Cachorro e Gato herda todas as características (atributos e métodos) presentes na classe Animal, e mais algumas características próprias.

Imagine que vamos criar um sistema de cadastro de animais. Vamos, por questões didáticas, supor que todos os animais de nosso sistema fiquem armazenados em um ***array***. Então deve ser criado um ***array*** contendo objetos

dos tipos Gato e Cachorro. Mas armazenar diversos tipos diferentes de objetos em um único ***array*** não é uma boa ideia, porque é bastante complicado extrair essas informações de volta. Mas pare e pense por um instante: objetos do tipo Cachorro e objetos do tipo Gato são também objetos do tipo Animal, correto? Bom, então podemos criar um ***array*** capaz de armazenar Animais! Assim, nossa vida fica bem mais fácil, bastando atribuir ao ***array*** os objetos (do tipo Cachorro e Gato - que estendem a classe Animal) que queremos guardar. Em forma algorítmica, seria mais ou menos:

listaDeAnimais = new Animal[100];

Criamos um ***array*** com 100 posições que armazena objetos do tipo Animal.

listaDeAnimais[0] = new Cachorro(“Dunga”);

Criamos um novo objeto do tipo Cachorro com o nome Dunga e armazenamos no ***array***.

listaDeAnimais[1] = new Gato(“Chico”);

Criamos um novo objeto do tipo Gato com o nome Chico e armazenamos no ***array***.

Certo! Agora temos um ***array*** com vários objetos do tipo Animal. Agora vamos fazer um looping por todos esses objetos, enviando para cada um deles a mensagem “fale”. O que iria acontecer?

Inicialmente, vamos supor que a classe abstrata Animal possua o método *“fale”*, e que ele seja implementado (de forma algorítmica) da seguinte forma:

```
Classe Animal {
método fale() {
imprimaNaTela(“ Eu sou mudo! “);
}
}
```

Desta forma, todo objeto de alguma classe que estenda a classe Animal vai ter, automaticamente, o método *“fale”,* e isso inclui todos os objetos das classes Cachorro e Gato. Mas, todos eles, ao receberem a mensagem *“fale”*, vão responder, imprimindo na tela a mensagem *“Eu sou mudo!”*. Mas Gatos e Cachorros podem falar! O que podemos fazer é sobrescrever o método fale para cada uma das classes, substituindo, então, seu conteúdo pelo comportamento que queremos que cada subclasse tenha.

Por exemplo, poderíamos escrever na classe Gato do seguinte método:

```
Classe Gato {
Método fale() {
imprimaNaTela(“ Miaaaaaauuuuuu! “);
}
}
```

Para a classe Cachorro, poderíamos fazer de forma semelhante:

```
Classe Cachorro {
Método fale() {
imprimaNaTela(“ Au au au! “);
}      }
```

Agora, se fizermos um *looping* entre todos os objetos contidos em nosso *array* criado anteriormente, enviando para cada objeto a mensagem *“fale”*, cada um deles irá ter um comportamento diferente, dependendo se é um Cachorro ou um Gato. Nosso *looping* entre todos os animais cadastrados no nosso sistema seria mais ou menos assim:

```
    int cont;
    para cont de 0 a 100 faça {
[cont].fale();
     }
```

Isto é polimorfismo! Uma mesma mensagem é enviada para diferentes objetos da mesma classe (Animal) e o resultado pode ser diferente, para cada caso.

EXERCÍCIO

1. O que realmente é um tipo abstrato de dados ?

2. Qual o papel do encapsulamento no que tange à confiabilidade de programas?

3. O que realmente significa herança, no contexto das linguagens de programação?

4. O que realmente é polimorfismo?

5. Qual a forma de vinculação dinâmica deve ser empregada pelos mecanismos de suporte à orientação a objetos?

6. Que problemas os tipos abstratos de dados vieram resolver?

7. O que são classes abstratas?

8. Qual a origem da linguagem Eiffel?

9. Como Eiffel suporta orientação a objetos?

10. Eiffel é uma linguagem puramente orientada a objetos? De que forma?

11. Que realmente significa passagem de mensagens?

12. Qual o objetivo da pseudo-variável super em Smalltalk?

13. Por que as classes em Smalltalk podem responder a mensagens?

14. Que é uma função amiga em C++?

15. Onde os objetos em C++ são alocados?

16. Todas as subclasses em Ada 95 são de que tipos ?

SAIBA MAIS

Existem muitos bons textos sobre este tema. Alguns deles estão listados na Referências colocada ao final desta Unidade. A nosso juízo, o livro de R. Sebesta apresenta um diferencial nesta área. Em particular ele dedica dois capítulos importantes a este tema.

Muito se tem publicado em nível de notas de aula e apostilas publicadas pela Wikipédia e outras formas de publicação, a maioria delas, disponíveis pela Internet.

Vale ressaltar que muitos manuais tratam do uso e não da Programação Orientada a Objetos, apesar da linguagem a que ele se refere incorporar este mecanismo.

Referências

AHO, A. V. R; ULLMAN, J. D. Compilers: principles, techniques, and tools. Reading, MA: Addison-Wesley, 1986.

DIJKSTRA, E.W. A discipline of programming. Englewood Cliffs, N. J: Prentice-Hall, 1976.

DIJKSTRA, E.W. Selected writings on computing: a personal perspective. New York : Spring-Verlag, 1982.

GHEZZI, C; JAZAYERI, M. Conceitos de linguagens de programação. Rio de Janeiro: Campus, 1985.

GOLDBERG, A; ROBSON, D. Smalltalk-80: the language and Its implementation. Reading, MA: Addison-Wesley, 1989.

GOSLING, J; JOY, B; STEELE, G. The Java language specification. Reading, MA: Addison-Wesley, 1996.

HOARE, C. A. R; WIRTH, N. An axiomatic definition of the programming language pascal. Acta Informatica. v. 2. p 335-355. 1973.

NETTO, José Lucas M. Rangel. Projeto de linguagens de programação. Notas de aula. COPPE, UFRJ. 1993.

NUNES, Maria das Graças Volpe; FACELI, Katti. Tópicos em linguagens de programação. Notas de aula. Instituto de Ciências Matemáticas e de Computação – USP.

O'SULLIVAN, Bryan; GOERZEN, John; STEWART, Don. Real world haskell. O'Reilly. 2008.

PAULSON, Laurence C. ML for the working programmer. New York: Cambridge University Press, 1991.

SCHMIDT, D. A. Denotational semantics. Massachusetts: Allyn and Bacon, 1986.

SCOTT, D. Data types as lattices. SIAM Journal of Computing. vol. 5,3. 1976.

SCOTT Michael L. Programming language pragmatics. Morgan Kaufmann Publishers. 2000.

SEBESTA, Robert W. Programming languages concepts. 7th. edition, New York: John Wiley & Sons. 2005.

SETHI, Ravi. Programming languages concepts and constructs. Addison Wesley Publishing Company. 1989.

SILVA, José Carlos G; ASSIS, Fidelis S. G. de. Linguagens de programação conceitos e avaliação. São Paulo: McGraw-Hill, 1988.

SKIENA, Steven S; REVILLA, Miguel A. Programming challenges: the programming context training manual. Texts in Computer Science. Springer Science+Business Media, Inc. 2003.

SOUZA, Francisco Vieira de; LINS, R. D. Analysing space behaviour of functional programs. Recife: Conferência Latino-americana de Programação Funcional, 1999.

SOUZA, Francisco Vieira de. Aspectos de eficiência em algoritmos para o gerenciamento automático dinâmico de Memória. Tese de Doutorado. Recife: Centro de Informática-UFPE, 2000.

STOY, J. E. Denotational semantics: the scott-strachey approach to programming language theory. MIT Press, 1977.

VELOSO, Paulo A S. Estruturação e verificação de programas com tipos de dados. São Paulo: Edgar Blücher, 1987.

WATT, David A. Programming language concepts and paradigms. 2nd ed. New York: Prentice Hall International, 2004.

REFERÊNCIAS NA WEB

www.ufpi.br/uapi (A Página da Universidade Aberta do Piauí - UAPI)
www.uab.gov.br (O Site da Universidade Aberta do Brasil- UAB)
www.seed.mec.gov.br (A Homepage da Secretaria de Educação a Distância do MEC - SEED)
www.abed.org.br (O site da Associação Brasileira de Educação a Distância - ABED)
http://pt.wikipedia.org/ O site da Wikipedia.
www.inf.ufsc.br/ine5365/introlog.html
www.gregosetroianos.mat.br/logica.asp.

www.uapi.ufpi.br