Capítulo I – Introdução

"O aspecto mais importante, mas também o mais elusivo, de qualquer ferramenta é sua influência nos hábitos daqueles que se treinam no seu uso. Se a ferramenta é uma linguagem de programação essa influência é, gostemos ou não, uma influência em nosso hábito de pensar."

Edsger W. Dijkstra

Uma Linguagem de Programação (LP) é um instrumento utilizado pelo profissional de computação para escrever programas, isto é, conjuntos de instruções a serem seguidas pelo computador para realizar um determinado processo.

Em virtude das limitações físicas dos computadores e da pouca maturidade da ciência da computação na época de surgimento dos primeiros computadores, eles só podiam ser programados através de linguagens de programação muito simples. Tais linguagens disponibilizavam um pequeno conjunto de tipos de instruções capazes de realizar ações muito elementares e se caracterizavam por serem de uso exclusivo de um computador específico. Por conta disso, hoje elas são conhecidas como linguagem de máquina ou de baixo nível.

À medida que a computação avançava e se vislumbrava o potencial dessa nova ferramenta, as aplicações iam se tornando cada vez mais complexas. Nessa época, foi constatado que o uso de linguagens tão simples e específicas reduzia significativamente a produtividade dos programadores e impedia a ampla disseminação dos computadores. Para contornar esse problema, surgiram as linguagens de programação de alto nível. Em contraste com as linguagens de máquina, essas linguagens se caracterizam por não serem específicas para um computador e por terem um conjunto mais amplo e expressivo de tipos de instrução.

O enfoque desse livro é no estudo dessas linguagens. Dessa maneira, sempre que for utilizado o termo "linguagem de programação", estaremos nos referindo a uma linguagem de programação de alto nível.

Para que os programadores possam construir programas em uma determinada linguagem é necessário definir os símbolos que podem ser utilizados e a forma como devem ser combinados para produzir um programa válido. Além disso, os programadores também precisam entender como um programa válido será executado pelo computador.

Embora existam inúmeras linguagens de programação, cada qual com seu próprio conjunto de símbolos e regras para formação e interpretação de programas, é possível e interessante estudá-las focando os principais con-

ceitos que lhes são comuns e esclarecendo como eles podem ser utilizados na programação e como podem ser implementados nas linguagens. Conceitos relevantes e específicos existentes em uma linguagem particular também devem ser compreendidos pois podem ser um instrumento valioso para melhor entendimento de uma técnica de programação ou para futuras evoluções e novos projetos de linguagens.

Para tornar o aprendizado de um conceito mais efetivo, é importante que ele seja utilizado na construção de um programa em uma linguagem de programação que ofereça recursos para utilização do conceito. As três linguagens (C, C++ e JAVA), que são usadas primordialmente nos exemplos deste texto, oferecem recursos para a programação da grande maioria dos conceitos abordados aqui.

O estudo e a discussão sobre as características e os mecanismos existentes em linguagens de programação requer entendimento sobre as propriedades desejáveis em uma linguagem de programação, os papéis que podem ser exercidos pela linguagem e o contexto no qual seu projeto foi realizado.

Nesse capítulo é apresentada uma introdução ao estudo de Linguagens de Programação. São discutidas as razões para se conhecer profundamente esse tema, o papel das linguagens de programação no processo de desenvolvimento de programas e as principais propriedades desejáveis em linguagens de programação. Em seguida, são apresentadas noções importantes sobre a especificação, implementação e padronização de uma linguagem. Cada um dos paradigmas mais comuns de linguagens de programação são brevemente abordados. Por fim, é discutida a evolução e história dessas linguagens.

1.1 Razões para Estudar Linguagens de Programação

Embora qualquer programador reconheça que linguagens de programação são instrumentos fundamentais dentro de sua especialidade, é importante destacar quais são os benefícios que um estudo aprofundado de Linguagens de Programação pode proporcionar ao estudante. Esses benefícios são apresentados a seguir:

a) Maior capacidade de desenvolver soluções computacionais para problemas. Uma maior compreensão sobre os conceitos de uma LP pode aumentar nossa habilidade em como pensar e resolver problemas. Por exemplo, o conhecimento do conceito de tipos abstratos de dados estimulam a utilização desse método de programação mesmo em LPs que não possuem mecanismos específicos para a sua implementação.

b) Maior habilidade ao usar uma LP. O maior entendimento a respeito das funcionalidades e implementação de uma LP possibilita ao programador construir programas melhores e mais eficientes. Por exemplo, conhecendo como as LPs são implementadas pode-se entender porque algoritmos recursivos são menos eficientes que os iterativos correspondentes.

c) Maior capacidade para escolher LPs apropriadas. Conhecer os recursos oferecidos por uma linguagem e saber como esses recursos são implementados pode determinar uma boa escolha da linguagem de programação a ser usada em um projeto. Por exemplo, saber que C não realiza checagem dinâmica dos índices de acessos a posições de vetores pode ser decisivo para escolher essa linguagem em aplicações de tempo real que fazem uso frequente de acessos vetoriais.

d) Maior habilidade para aprender novas LPs. Por exemplo, programadores que aprenderam os conceitos de orientação a objetos tem maior facilidade para aprender C++ e JAVA.

e) Maior habilidade para projetar novas LPs. Muito embora poucos profissionais de computação tenham a oportunidade, ao longo de suas carreiras, de participar da criação de uma linguagem de programação de propósito geral, não é raro se depararem com situações nas quais é necessário projetar linguagens para um propósito específico. Por exemplo, ao construir as interfaces com o usuário de sistemas pode ser necessário projetar e implementar uma linguagem de comandos para comunicação entre o usuário e o sistema.

1.2 O Papel das Linguagens de Programação no Processo de Desenvolvimento de Software

Linguagens de programação foram criadas para tornar mais produtivo o trabalho dos programadores. Logo, em última instância, o objetivo das linguagens de programação é tornar mais efetivo o processo de desenvolvimento de software. É importante enfatizar que esse processo existe tanto para tornar mais produtiva a geração e manutenção de software quanto para garantir que ele seja produzido atendendo a padrões de qualidade.

Assim, uma maneira de saber como as linguagens de programação podem apoiar esse processo envolve o conhecimento das propriedades requeridas em um software de qualidade. As principais propriedades desejadas em um software são confiabilidade, manutenibilidade e eficiência.

A confiabilidade diz respeito ao atendimento adequado da especificação funcional, da garantia de segurança contra erros e da integridade dos dados manipulados pelo software. Uma LP pode promover a confiabilidade

de programas facilitando a existência de ferramentas computacionais que verifiquem a ocorrência de erros nos programas. Por exemplo, LPs que requerem a declaração de variáveis, tais como C, PASCAL, e MODULA-2, facilitam a identificação de erros de digitação de nomes. Caso um usuário digite um nome incorreto, um verificador de erros pode identificá-lo porque não foi declarado.

A manutenibilidade diz respeito a facilidade de alteração do software. Necessidades de modificação são provenientes de erros de especificação, projeto ou implementação, de alterações no ambiente tecnológico onde o software é executado e de novas demandas do usuário. Uma LP pode promover a manutenibilidade de programas fornecendo mecanismos que permitam a sua adaptação a diferentes contextos. Por exemplo, a declaração de constantes em Pascal e Modula-2 facilita a realização de modificações nos programas. Caso um programa utilize uma constante para definir o tamanho máximo de um vetor, basta modificar essa constante para adaptar todo o programa a um aumento no tamanho máximo do vetor.

A eficiência diz respeito ao uso otimizado dos recursos computacionais em termos de tempo de execução, de espaço de memória utilizado e de uso de dispositivos periféricos. Uma LP pode promover a eficiência de programas incentivando o uso de mecanismos computacionalmente eficientes. Por exemplo, FORTRAN (com exceção de FORTRAN 90) não permite o uso de recursão para tornar mais eficiente o processamento e o consumo de memória.

Outro modo de saber como as linguagens de programação podem apoiar o processo de desenvolvimento de software é conhecendo como ele é realizado. O processo de desenvolvimento de software é normalmente compreendido em cinco etapas [PRESSMAN, 1997]:

1.2.1 Especificação de Requisitos

Nessa etapa se identifica o que o software deverá realizar em termos de funcionalidades. É necessário especificar o ambiente no qual o software atuará, quais serão as suas atividades, quais os impactos que deverá produzir e de que maneira ele deverá interagir com os usuários. Em particular, dever-se-á especificar os requisitos de desempenho do sistema em termos de tempo de resposta desejado, espaço de memória requerido e também de interação com outros dispositivos e usuários.

Outra atividade importante nessa etapa é a realização de um estudo de viabilidade e custo do software a ser desenvolvido. Esse estudo tem por objetivo responder se a confecção e implantação do software é viável técnica, cronológica e socialmente, bem como determinar, através da estimativa do custo de desenvolvimento do software, se a sua construção é viá-

vel economicamente. A confecção de versões dos manuais do usuário do software também é uma atividade desta etapa.

O estudo de linguagens de programação influencia pouco a realização dessa etapa. Basicamente, o conhecimento sobre linguagens de programação pode ser usado no estudo de viabilidade para ajudar a responder se é possível desenvolver o software no período de tempo desejado.

1.2.2 Projeto do Software

Tendo por base os documentos de especificação de requisitos, pode-se projetar o sistema de programação. O projeto envolve a escolha de um método de projeto e sua aplicação na especificação da arquitetura do software e seus procedimentos, das suas estruturas de dados e de suas interfaces.

O resultado desta fase é um documento de especificação do projeto do sistema, identificando todos os módulos que compõem a arquitetura do sistema, as estruturas de dados utilizadas por cada módulo, as interfaces de comunicação entre módulos (interfaces internas), com outros sistemas (interfaces externas) e com as pessoas que o utilizam (interface com o usuário). Também faz parte desse documento as descrições procedimentais de cada módulo da arquitetura.

O principal papel das LPs nessa etapa é dar suporte ao método de projeto. Isto possibilita que a implementação de um sistema reflita o seu projeto, evitando adaptações e distorções no projeto e perda de correspondência. Pode-se observar que algumas linguagens de programação são mais adequadas quando se utilizam certos métodos de projeto. Por exemplo, enquanto C é mais adequada ao método de projeto hierárquico-funcional, JAVA é mais adequada ao método orientado a objetos.

Existem várias ferramentas CASE que oferecem suporte às atividades dessa etapa. Muitas dessas ferramentas já geram parte da codificação do software em uma LP.

1.2.3 Implementação

A etapa de implementação é onde ocorre a programação dos módulos do software. Obviamente, LPs são essenciais nessa etapa uma vez que os programas devem ser escritos em uma linguagem. Essa etapa é a mais atendida por ferramentas, tais como editores de texto que destacam os vocábulos da linguagem e identam automaticamente o texto, analisadores léxicos, sintáticos e semânticos de programas e bibliotecas de subprogramas e módulos.

1.2.4 Validação

O propósito dessa etapa é verificar se o sistema satisfaz as exigências das especificações de requisitos e de projeto. Geralmente, isto é feito testando-se o sistema contra as especificações. Existem três tipos de testes: teste de módulo, teste de integração e teste de sistema. No teste de módulo é verificado se ele cumpre o que lhe foi especificado. No teste de integração é verificado se os módulos se integram apropriadamente, isto é, se eles interagem tal como estabelecido nas especificações de suas interfaces. O teste de sistema averigua se o software cumpre as funcionalidades para o qual foi desenvolvido e atende a todos os demais requisitos de usabilidade e eficiência.

LPs podem auxiliar a validação de vários modos. Por exemplo, a natureza de algumas linguagens facilita a construção de depuradores de erros e ambientes nos quais é fácil executar módulos de programa independentemente da existência de outros módulos. Isso pode ser muito valioso para realizar os testes de módulo e integração.

1.2.5 Manutenção

A última etapa do processo de desenvolvimento de software é a manutenção e evolução do sistema. Para que o ciclo de vida de um software possa ser ampliado é necessário que ele seja capaz de facilitar a correção de erros residuais (isto é, erros descobertos após a sua liberação para o usuário), adaptar-se a mudanças no seu contexto de aplicação (tal como, um novo ambiente computacional) e atender a demandas por melhoria e inclusão de serviços.

LPs que oferecem recursos de modularização tendem a gerar programas mais fáceis de serem mantidos uma vez que as alterações em um módulo não interferem nos demais módulos constituintes do software.

1.3 Propriedades Desejáveis em uma Linguagem de Programação

A partir da chamada crise do software [PRESSMAN, 1997], o aproveitamento do tempo do profissional de programação se tornou um conceito central no processo de desenvolvimento de software. Consequentemente, as propriedades desejáveis nas LPs devem enfatizar esse aspecto. Discutem-se a seguir algumas das principais propriedades desejáveis em uma LP:

1.3.1 Legibilidade

Essa propriedade diz respeito à facilidade para se ler e entender um programa. Quanto mais fácil for seguir as instruções de um programa, mais fácil será entender o que está sendo feito e também descobrir erros de programação.

LPs que requerem o uso extensivo do comando *goto* normalmente reduzem a legibilidade dos programas porque permitem a ocorrência da chamada programação macarrônica ou não estruturada. Nesse tipo de programação, os programas possuem fluxo de controle que não obedecem a padrões regulares. Isto torna difícil acompanhar e entender o que fazem esses programas. Programas em versões antigas de FORTRAN e BASIC, por exemplo, tendiam a ser mal estruturados porque estas versões requeriam o uso de *goto* para implementar as estruturas de seleção e repetição.

O uso de um mesmo vocábulo da LP para denotar diferentes comportamentos dependendo do contexto onde é usado também é prejudicial à legibilidade. Por exemplo, o vocábulo *this* pode ser usado em JAVA tanto para referenciar um objeto quanto para fazer uma chamada a uma função construtora de dentro de outra. Outro exemplo é o operador * em C, que tanto pode denotar a operação de multiplicação de números quanto operações de manipulação de ponteiros. Isso permite a criação de expressões confusas. Por exemplo, a seguinte linha de código C apresenta o uso do operador * em três diferentes contextos:

```
*p = (*p)*q;
```

Observe que a operação designada pelo * mais a direita nessa linha de código é a operação de multiplicação. Por sua vez, a operação designada pelo * do meio é a de retorno do conteúdo da célula de memória apontada pelo ponteiro *p*. Por fim, o * mais à esquerda denota a operação de retorno do endereço da célula apontada por *p*.

Efeitos colaterais são mudanças adicionais promovidas no estado do programa (isto é, nos valores das variáveis do programa) durante a avaliação de uma determinada de expressão ou a execução de um comando ou subprograma. O termo adicionais se refere ao fato das mudanças provocadas pelos efeitos colaterais não serem o objetivo principal da expressão, comando ou subprograma realizado. Por exemplo, a operação *x*++ de C tem como efeito principal retornar o valor da variável *x* e como efeito colateral incrementar o valor dessa variável. Efeitos colaterais podem ser prejudiciais a legibilidade quando seus resultados não ficam explícitos no trecho de programa que utiliza a operação. Observe no exemplo 1.1, em C, que a chamada da função *retornaCinco* provoca alteração na variável global *x*.

```
int x = 1;
int retornaCinco() {
  x = x + 3;
 return 5;
}
main() {
  int y;
  y = retornaCinco ();
  y = y + x;
}
```

Exemplo 1. 1- Problema de Legibilidade Causado por Efeito Colateral em C

Note que esse efeito não fica explícito no trecho de código que chama *retornaCinco*. Alguém que fizesse uma rápida inspeção no código do exemplo 1.1 tentando identificar o que o programa faz e apenas olhasse a função *main* não conseguiria entender que o valor final de *y* é nove e não seis. Efeitos colaterais podem causar problemas ainda mais graves do que o de legibilidade (por exemplo, podem causar indeterminismo na expressão *x + retornaCinco()*)[1.1].

Marcadores de blocos de comandos, tais como o *BEGIN*(*{*)–*END*(*}*) de PASCAL e C, também podem causar confusões na leitura do programa quando existem vários comandos de repetição e seleção aninhados. A inexistência de um marcador específico que indique onde o comando *if* de C se encerra possibilita a escrita de comandos ifs aninhados difíceis de serem entendidos. No exemplo 1.2, embora o *else* pertença ao *if* mais interno, tem-se a impressão que ele se refere ao *if* mais externo.

```
if (x>1)
        if (x==2)
                x=3;
else
        x=4;
```

Exemplo 1. 2 – Problema de Legibilidade Relacionado com Marcadores de Bloco

ADA reduz este problema usando *begin–endif* e *begin–endloop*.

Algumas LPs adotaram posturas altamente questionáveis com relação à legibilidade. FORTRAN, por exemplo, permite que palavras reservadas como *DO*, *END, INTEGER* e *REAL* sejam também nomes de variáveis.

[1.1] Esse problema será discutido em maiores detalhes no capítulo 5 desse livro.

1.3.2 Redigibilidade

Essa propriedade possibilita ao programador se concentrar nos algoritmos centrais do programa, sem se preocupar com aspectos não relevantes para a resolução do problema. Esta característica é a que melhor diferencia as linguagens de máquina (nas quais o programador deve se preocupar principalmente com detalhes de implementação) e linguagens de programação (nas quais o programador se concentra na descrição do algoritmo que resolve o problema).

LPs com tipos de dados limitados requerem o uso de estruturas complexas. Isto acaba dificultando a redação de programas. Por exemplo, como FORTRAN não possui registros, armazenar dados de empregados de uma firma requer a criação de vetores específicos para cada tipo de dado. Ao redigir um subprograma para ordenar os dados seria necessário usar várias instruções para trocar os elementos correspondentes em cada vetor.

LPs que requerem muita programação de entrada e saída e que não dispõem de mecanismos para o tratamento de erros tendem a obscurecer os algoritmos centrais nos programas.

A redigibilidade de programas pode conflitar com a legibilidade. C permite a redação de comandos complexos, mas que podem não identificar de maneira muito clara a sua funcionalidade. Observe o comando *for* do exemplo 1.3 e tente identificar o que ele faz.

```
void f(char *q, char *p) {
   for  (;*q=*p; q++,p++);
}
```

Exemplo 1. 3 - Redigibilidade X Legibilidade

É fácil perceber o quão concisa é essa implementação. Contudo, o preço a ser pago é a falta de entendimento imediato sobre sua funcionalidade. Programadores inexperientes encontrarão dificuldades para entender o que esse comando faz, enquanto programadores experientes poderão se confundir ao fazer uma rápida leitura.

1.3.3 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 compilação ou execução. Um compilador de JAVA pode detectar durante a compilação um erro de digitação cometido

por um programador em situações como a do exemplo 1.4, onde o *v* foi inadvertidamente trocado por *u* no trecho *v = u + 2;* :

```
boolean u = true;
int v = 0;
while (u && v < 9) {
  v = u + 2;
  if (v == 6) u = false;
}
```

Exemplo 1. 4 – Declaração de Tipos e Confiabilidade

LPs que possuem mecanismos para detectar eventos indesejáveis e especificar respostas adequadas a tais eventos permitem a construção de programas mais confiáveis. No trecho de código em JAVA, apresentado no exemplo 1.5, poderá haver um erro se o valor de *i* estiver fora dos limites de índices do vetor *a*. Caso isto ocorra, o programa interrompe seu fluxo normal de execução e passa para o trecho de código responsável pelo tratamento desse erro.

```
try {
   System.out.println(a[i]);
} catch (IndexOutofBoundsException) {
   System.out.println("Erro de Indexação");
}
```

Exemplo 1. 5 - Tratamento de Exceções e Confiabilidade

1.3.4 Eficiência

De acordo com as demandas por recursos de um tipo de aplicação, certas LPs são mais recomendadas e outras não devem ser usadas. Aplicações 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 espaço de memória.

Muito embora, hoje, boa parte da responsabilidade em gerar código eficiente seja transferida para o compilador, através da otimização automática de código, as características de uma LP podem determinar se o programa gerado naquela LP será mais ou menos eficiente. Assim, LPs que requerem a verificação de tipos durante a execução são menos eficientes do que aquelas que não fazem este tipo de requisição.

Por exemplo, o mecanismo de tratamento de exceções existente em JAVA impõem que os índices de vetores sejam verificados em todos os acessos durante a execução dos programas. Isso implica na necessidade de se fazer um teste antes de qualquer acesso aos vetores. Por outro lado,

como C não faz esse tipo de exigência, o código gerado economizará a realização desse teste e, portanto, será mais veloz.

1.3.5 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, tal como C++, tendem a ser mais difíceis de aprender. Por exemplo, num determinado contexto, os seguintes comandos em C ou C++, têm o mesmo efeito:

```
c = c + 1;        c+=1;        c++;        ++c;
```

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.

1.3.6 Ortogonalidade

Ortogonalidade 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 número 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 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, por sua vez, dificulta o aprendizado da LP e pode estimular a ocorrência de erros de programação. No exemplo 1.6 é mostrada a falta de ortogonalidade de um código JAVA:

```
int  x, y = 2, z = 3;
byte a, b = 2, c = 3;
x = y + z;
a = b + c;
```

Exemplo 1. 6 - Falta de Ortogonalidade em JAVA

Embora tanto o tipo *int* quanto o tipo *byte* sejam tipos inteiros, a linha de código onde ocorre a soma de tipos *int* é legal enquanto a que soma tipos *byte* é ilegal (tente descobrir porquê). Ora, essa falta de ortogonalidade claramente é uma fonte potencial de erros, uma vez que a maioria dos

programadores pensaria que a mesma regra que se aplica ao tipo *int* se aplicaria aos outros tipos de inteiros.

Outro exemplo clássico de falta de ortogonalidade ocorre em PASCAL. Nessa LP, funções podem retornar qualquer tipo de dados com exceção de registros e vetores.

1.3.7 Reusabilidade

Outra propriedade desejável em LPs é a reusabilidade de código, isto é, a possibilidade de reutilizar o 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 novamente a partir do zero.

LPs podem incentivar a criação de código reusável de várias maneiras. A forma mais simples de facilitar a reusabilidade é através da parametrização de subprogramas. Por exemplo, o subprograma em C apresentado no exemplo 1.7 pode ser utilizado em qualquer aplicação na qual se queira trocar os valores de duas variáveis inteiras quaisquer.

```
void troca (int *x, int *y) {
  int z = *x;
  *x = *y;
  *y = z;
}
```

Exemplo 1. 7 - Reuso por Parametrização de Subprogramas

Outro mecanismo muito útil para permitir o reuso de código é a modularização através das bibliotecas de subprogramas. A linguagem C oferece inúmeras funções de entrada e saída (tais como, *printf*, *scanf* e *fprintf*) como parte de sua biblioteca padrão. Essas funções podem ser usadas em qualquer programa sem que o programador necessite reescrevê-las.

1.3.8 Modificabilidade

Essa propriedade se refere às facilidades oferecidas pelas LPs para possibilitar ao programador alterar o programa em função de novos requisitos sem que tais modificações impliquem em mudanças em outras partes do programa.

Exemplos 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 abstratos de dados.

Em C, várias constantes simbólicas são usadas, tais como *NULL* e *EOF*. O comando seguinte cria uma constante simbólica em C denotando o número *pi* com precisão de duas casas decimais.

```
const float pi = 3.14;
```

Se em algum momento for constatada a necessidade de maior precisão na definição do número *pi*, basta fazer a alteração nessa mesma linha (incluindo mais casas decimais) e todas as ocorrências da constante *pi* no programa serão ajustadas para o seu novo valor numérico, sem que seja necessário realizar alterações em outras partes do programa onde *pi* é usado.

1.3.9 Portabilidade

É altamente desejável que programas escritos em uma LP se comportem da mesma maneira independentemente da ferramenta utilizada para traduzir os programas 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 em diferentes situações sem que seja necessário dispender tempo de programação para reescrevê-los ou adaptá-los ao novo ambiente de tradução ou execução.

LPs podem facilitar a obtenção de programas portáveis através da amarração rigorosa do comportamento de seus elementos em tempo de projeto da linguagem, não dando liberdade para que os implementadores definam comportamentos distintos para um mesmo elemento.

Contudo, essa postura pode impor algumas restrições à implementação das linguagens, em particular, no que diz respeito à busca por eficiência na execução dos programas. Nesse caso, pode-se optar por sacrificar a completa portabilidade dos programas na LP em benefício da potencialização de outras propriedades. Mesmo assim, deve-se procurar maximizar a portabilidade permitindo que os programas escritos na LP sejam transportados para outros ambientes requerendo apenas poucas modificações em seu código.

1.4 Especificação de LPs

Ao se criar uma LP é necessário definir como se faz para escrever programas nessa linguagem e como os programas válidos devem se comportar. Essa definição deve ser feita através de documentos descritivos que estabeleçam de maneira precisa como essas duas atividades devem ser realizadas. Tais documentos formam a especificação da LP. Sem uma es-

pecificação apropriada, implementações das LPs não podem ter uniformidade, fazendo com que programas construídos para uma implementação tenham comportamento bem diferenciado ou mesmo não sejam válidos em outra implementação.

A especificação de uma LP requer a descrição de um léxico, de uma sintaxe e de uma semântica para a LP. O léxico da LP corresponde ao vocabulário que pode ser utilizado para formar sentenças na linguagem.

A sintaxe da LP corresponde ao conjunto de regras que determinam quais sentenças podem ser formadas a partir da combinação dos itens léxicos. O léxico e sintaxe estão relacionados com a forma dos programas, isto é, como expressões, comandos, declarações e outros elementos da LP podem ser combinados para formar programas válidos.

A semântica da LP descreve como as construções sintaticamente corretas são interpretadas ou executadas. A semântica está relacionada com o significado dos programas, isto é, como eles se comportam quando executados por computadores. Por exemplo, no comando seguinte, em C:

a = b;

- O léxico da LP estabelece que *a*, =, *b* e *;* fazem parte do vocabulário da LP.
- A sintaxe da LP indica que a sentença formada pelo identificador *a*, o símbolo =, o identificador *b* e o símbolo *;* designa um comando válido de atribuição.
- A semântica da LP indica que este comando deve ser executado de modo a substituir o valor de *a* pelo valor atual de *b*.

A sintaxe de uma LP influencia como os programas são escritos pelo programador, lidos por outros programadores e analisados pelo computador. A semântica de uma LP influencia como os programas são criados pelo programador, entendidos por outros programadores e interpretados pelo computador.

A sintaxe de uma LP é descrita por uma gramática. Uma notação muito utilizada para descrever gramáticas de LPs é a BNF (Backus-Naur Form). O exemplo 1.8 apresenta uma gramática para formação de expressões aritméticas elementares, descrita em BNF:

<expressão> ::= <valor>|<valor><operador><expressão>
<valor> ::= <número>|<sinal><número>
<número> ::= <semsinal>|<semsinal>.<semsinal>
<semsinal> ::= <dígito>|<dígito><semsinal>
<dígito> ::= 0 |1 |2 |3 |4 |5 |6 |7 |8 |9

<sinal> ::= + | –
*<operador> ::= + | – | / | *

Exemplo 1. 8 - Gramática Simples

Recomenda-se descrever a semântica de uma LP formalmente. Contudo, descrições formais de semântica são freqüentemente complexas para ler e escrever. Como resultado, na maior parte das vezes, a semântica de LPs é descrita de maneira informal através de documentos que explicam em linguagem natural qual o significado dos comandos da LP. Uma outra abordagem muito adotada é o enfoque operacional, que consiste em descrever o significado de um comando através da apresentação de um código equivalente numa linguagem mais elementar.

Um problema que ocorre muito frequentemente com LPs é a ausência de uma padronização. Ao invés de se ter uma especificação única adotada por todos os implementadores de LPs, surgem várias implementações distintas com as suas próprias especificações. Isso ocorre normalmente quando a linguagem ainda não se encontra totalmente estabelecida. Assim, os implementadores discordam a respeito de quais elementos devem fazer parte da linguagem e como eles devem se comportar. Muitas vezes essa discordância é provocada pela necessidade de explorar as características específicas do ambiente onde a implementação será realizada.

Contudo, na medida que a linguagem vai se popularizando e amadurecendo, essa variação de comportamentos acaba por gerar problemas significativos de portabilidade de programas.

Para resolver esses problemas, procura-se estabelecer uma especificação padrão única que deve ser respeitada por todos os implementadores da LP. Variações podem até continuar a existir, mas um núcleo de elementos comum necessita ser implementado para que a LP esteja em conformidadade com o padrão estabelecido. Assim, os programadores podem escrever programas garantidamente portáveis, desde que só utilizem os elementos padronizados.

Normalmente, a padronização de uma LP é promovida por alguma instituição especializada nesse serviço, tais como a ISO (International Standards Organization), o IEEE (Institute of Electrical and Electronics Engineers), o ANSI (American National Standards Institute) e o NIST (National Institute for Standards and Technology). O processo de padronização envolve a formação de um grupo de voluntários especialistas que trabalham para definir quais elementos devem fazer parte da padronização.

Esse trabalho é complexo e demorado, pois envolve a obtenção de consenso entre os participantes do grupo de padronização. Além disso, o consenso é geralmente obtido omitindo-se as características mais polêmicas.

Outro problema é a definição do momento de padronização. Se muito cedo, a falta de experiência com a LP pode produzir um padrão inadequado que iniba o seu uso e disseminação. Se muito tarde, a existência de muitas versões incompatíveis com um grande legado de código pode dificultar ou retardar a aceitação do padrão pela comunidade.

1.5 Métodos de Implementação de LPs

Todo e qualquer programa escrito em uma LP necessita ser traduzido para a linguagem de máquina para ser executado. Para fazer isto é necessário aplicar um programa (ou conjunto de programas) que receba como entrada o código fonte do programa a ser traduzido e gere o código traduzido na linguagem de máquina. Esse programa tradutor é quem determina como os programas na LP serão implementados, isto é, como o código fonte traduzido se comportará efetivamente quando executado no computador. Sebesta [SEBESTA, 1999] descreve três métodos gerais de implementação de LPs (ilustrados na figura 1.1).

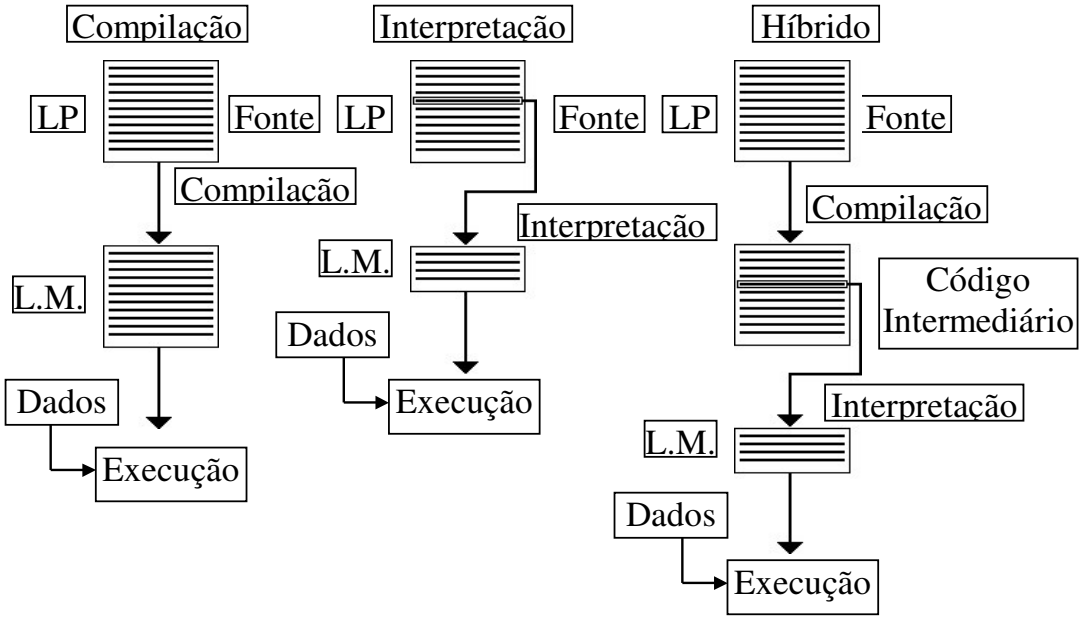

Figura 1. 1 - Métodos de Implementação de LPs

1.5.1 Compilação

O processo de compilação efetua a tradução integral do programa fonte para o código de máquina. Uma vez traduzido, o programa em linguagem de máquina pode ser executado diretamente.

A grande vantagem desse método de tradução é a otimização da eficiência na execução dos programas. A execução é rápida porque não se ne-

cessita fazer qualquer tradução durante a execução e porque boa parte das verificações de erros já podem ser efetuadas durante a tradução. Além disso, o próprio tradutor tem mais liberdade para realizar otimizações na geração do código executável, uma vez que pode considerar o código fonte globalmente.

Uma outra vantagem do processo de compilação é requerer apenas o código executável para que o programa possa ser executado. Assim, não é necessário possuir o código fonte do programa para sua execução.

A principal desvantagem do método de compilação é a não portabilidade do código executável para máquinas com arquitetura diferenciada daquela na qual ele foi compilado. Por exemplo, um programa em C compilado sobre o ambiente Linux não é executado sobre o ambiente Windows e vice-versa.

Outro problema se refere à depuração. Uma vez que o código executável normalmente não guarda referências ao texto do código fonte, qualquer indicação de erro de execução não pode ser devidamente identificada com a informação do nome da variável envolvida no erro ou da linha correspondente no código fonte que ocasiona o erro. É importante mencionar, no entanto, que existem sistemas de desenvolvimento de programas que permitem a depuração de programas compilados. Esses sistemas mantêm uma correspondência entre o código compilado e o código fonte para poder realizar a execução passo a passo do código, inspeção de valores de variáveis durante a execução e prestar informações referentes ao código fonte sobre um eventual erro de execução.

1.5.2 Interpretação pura

No processo de interpretação pura, um programa interpretador age como um simulador de um computador virtual que entende as instruções da LP. Basicamente, cada instrução do código fonte é traduzida para a linguagem de máquina quando essa instrução necessita ser executada. Imediatamente após a tradução, o código gerado é executado. Deste modo, em contraste com a compilação, a tradução e a execução de programas interpretados podem ser vistos como um processo único, não existindo etapas separadas de tradução e de execução.

A interpretação pura apresenta como vantagens: a facilidade para prototipação, visto que se pode executar comandos e partes do programa assim que são construídos, verificando se atuam corretamente; a facilidade de depuração, visto que as mensagens de erro podem se referir diretamente ao código fonte; e a facilidade de escrever programas mais flexíveis, visto que o interpretador da LP está presente durante a execução (permitindo,

por exemplo, a execução de trechos de programas criados, alterados ou obtidos durante a própria execução).

A grande desvantagem da interpretação pura em relação à compilação é a execução muito mais lenta do programa. A razão para essa lentidão decorre da necessidade do interpretador decodificar comandos complexos da LP, verificar erros do programa e gerar código em linguagem de máquina durante a própria execução do programa. Além disso, enquanto na compilação os comandos internos de uma repetição só necessitam ser traduzidos e verificados uma única vez, na interpretação pura, de modo geral, esse processo se repete para cada ciclo de execução da repetição.

Outra desvantagem é o maior consumo de espaço de memória, pois devem ser mantidos em memória uma tabela de símbolos, o código fonte e o próprio programa interpretador.

1.5.3 Híbrido

Processo que combina tanto a execução eficiente quanto a portabilidade de programas através da aplicação combinada dos dois métodos anteriores. A base para o método híbrido é a existência de um código intermediário, mais fácil de ser interpretado e, ao mesmo tempo, não específico de uma plataforma computacional. O método híbrido é dividido, portanto, em duas etapas: compilação para um código intermediário e interpretação deste código.

Embora ainda de execução mais lenta que o código compilado, a interpretação do código intermediário é muito mais rápida que a interpretação pura do código fonte porque as instruções do código intermediário são muito mais simples que as do código fonte e porque a maior parte das verificações de erro é realizada já na etapa de compilação.

Por sua vez, como o código intermediário não é específico para uma plataforma, os programas já compilados para este código podem ser portados para as mais diferentes plataformas sem necessidade de adaptação ou mesmo recompilação, bastando que exista um interpretador do código intermediário instalado na plataforma onde se deseja executar o programa.

JAVA adota o método híbrido. O código intermediário é chamado de bytecode. O interpretador de bytecode é a JVM (JAVA Virtual Machine). Cada plataforma computacional necessita possuir a sua própria JVM para que o programa em bytecode possa ser executado.

1.6 Paradigmas de LPs

Dá-se o nome de paradigma a um conjunto de características que servem para categorizar um grupo de linguagens. Existem diversas classificações de paradigmas de LPs, sendo a mais comum a que divide os paradigmas de LPs nos paradigmas imperativo, orientado a objetos, funcional e lógico. A classificação utilizada aqui adapta a proposta apresentada por Appleby [APPLEBY, 1991]. As únicas alterações realizadas nessa classificação são a substituição do termo distribuído pelo termo, mais genérico, concorrente e a remoção do paradigma de linguagens de bancos de dados. A figura 1.2 ilustra a classificação adotada aqui:

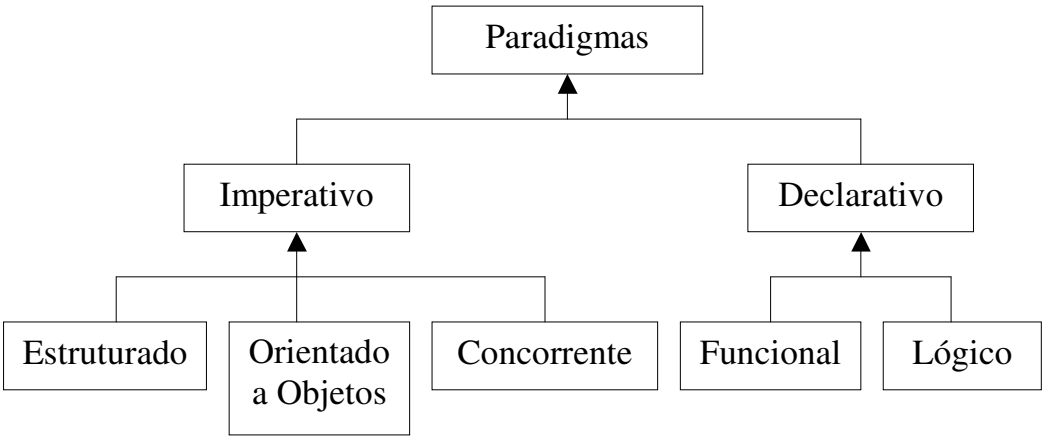

Figura 1. 2 - Paradigmas de LPs

Nessa classificação os paradigmas são subdivididos em duas categorias principais: imperativo e declarativo.

1.6.1 Paradigma Imperativo

O paradigma imperativo engloba os paradigmas baseados na idéia de computação como um processo que realiza mudanças de estados. Nesse sentido, um estado representa uma configuração qualquer da memória do computador. Programas de LPs que são incluídas nesse paradigma especificam como uma computação é realizada através de uma sequência de alterações no estado da memória do computador.

O foco dos programas no paradigma imperativo se encontra em especificar como um processamento deve ser feito no computador. Os conceitos fundamentais são de variável, valor e atribuição. Variáveis são vistas como sendo um conjunto de células de memória. Elas possuem um valor associado em um determinado instante do processamento e podem ter seu valor modificado através de operações de atribuição.

O paradigma imperativo é subdividido nos paradigmas estruturado, orientado a objetos e concorrente.

1.6.1.1 O Paradigma Estruturado

As primeiras LPs foram fortemente influenciadas pela programação em linguagem de máquina. Esse tipo de programação se caracterizava pela existência de uma sequência monolítica de comandos e pelo uso de desvios condicionais e incondicionais para determinar o fluxo de controle da execução do programa.

Logo se percebeu que esse estilo de programação estimulava a ocorrência de erros e reduzia a produtividade do programador. Para contornar essa dificuldade surgiu a programação estruturada. Esse tipo de programação se baseia na idéia de desenvolvimento de programas por refinamentos sucessivos (`top-down`). A programação estruturada consegue organizar o fluxo de controle de execução dos programas desestimulando o uso de comandos de desvio incondicional e incentivando a divisão dos programas em subprogramas e em blocos aninhados de comandos. PASCAL e C são linguagens que adotam o paradigma estruturado.

1.6.1.2 O Paradigma Orientado a Objetos

Com o avanço da computação, os sistemas de software têm se tornado cada vez maiores e mais complexos. O paradigma orientado a objetos oferece conceitos que objetivam tornar mais rápido e confiável o desenvolvimento desses sistemas.

Enquanto as linguagens que adotam o paradigma estruturado enfocam as abstrações de controle de execução dos programas, as linguagens que adotam o paradigma orientado a objetos enfocam as abstrações de dados como elemento básico de programação. Classes são abstrações que definem uma estrutura de dados e um conjunto de operações que podem ser realizadas sobre elas, chamadas métodos. Objetos são instâncias de classes. Outros conceitos importantes nesse paradigma são a herança e o polimorfismo.

Por utilizarem os conceitos do paradigma estruturado na especificação dos métodos, o paradigma orientado a objetos pode ser considerado uma evolução do paradigma estruturado. SMALLTALK, C++ e JAVA são linguagens que adotam o paradigma orientado a objetos.

1.6.1.3 O Paradigma Concorrente

A programação concorrente ocorre quando vários processos executam simultaneamente e concorrem por recursos. Sistemas concorrentes têm se tornado cada vez mais usados. Eles podem utilizar uma única unidade de processamento ou várias unidades em paralelo. Nesse último caso as unidades de processamento podem estar localizadas em um mesmo computador ou distribuídas entre vários. Sistemas concorrentes também podem compartilhar dados ou dispositivos periféricos.

O paradigma concorrente engloba linguagens que oferecem facilidades para o desenvolvimento desses sistemas. ADA é provavelmente a linguagem mais conhecida que oferece suporte a concorrência.

1.6.2 Paradigma Declarativo

Em contraste com o paradigma imperativo, no qual os programas são especificações de como o computador deve realizar uma tarefa, no paradigma declarativo os programas são especificações sobre o que é esta tarefa. No paradigma declarativo, o programador não precisa se preocupar sobre como o computador é implementado, nem sobre a maneira pela qual ele é melhor utilizado para realizar uma tarefa. A preocupação do programador é em descrever de forma abstrata a tarefa a ser resolvida.

Tipicamente, programas em linguagens declarativas são especificações de relações ou funções. Não existem atribuições a variáveis dos programas uma vez que as variáveis declarativas são de fato incógnitas e não representam células de memória.

Os interpretadores ou compiladores das LPs declarativas gerenciam a memória do computador, tornando transparente para o programador a necessidade de alocação e desalocação de memória.

1.6.2.1 O Paradigma Funcional

Linguagens funcionais operam apenas sobre funções, as quais recebem listas de valores e retornam um valor. O objetivo da programação funcional é definir uma função que retorne um valor como a resposta do problema. Um programa funcional é uma chamada de função que normalmente chama outras funções para gerar um valor de retorno. As principais operações nesse tipo de programação são a composição de funções e a chamada recursiva de funções. Outra característica importante é que funções são valores de primeira classe que podem ser passados para outras

funções. LISP, HASKELL e ML[1.2] são exemplos de linguagens funcionais.

1.6.2.2 O Paradigma Lógico

Linguagens lógicas são normalmente baseadas em um subconjunto do cálculo de predicados. Um predicado define uma relação entre fatos ou entre variáveis. Um programa lógico é composto por cláusulas que definem predicados e relações factuais. A característica diferencial do paradigma lógico é que a execução dos programas corresponde a um processo de dedução automática.

Assim, quando uma questão é formulada, um mecanismo de inferência tenta deduzir novos fatos a partir dos existentes para verificar a veracidade da questão. PROLOG é o exemplo mais conhecido de linguagem que adota o paradigma lógico.

1.7 Evolução das LPs

Antes do surgimento das LPs, a programação de computadores era feita exclusivamente em linguagem de máquina. Programadores tinham de conhecer profundamente a arquitetura da máquina onde o programa seria executado, seu conjunto de instruções e sua forma de funcionamento. Mesmo dominando todo esse conhecimento, a atividade de programação era pouco produtiva porque as instruções das linguagens de máquina são muito simples.

As primeiras LPs surgiram no final dos anos 50 e início dos anos 60 para facilitar o trabalho de programação. Por conta da cultura de programação dessa época e da limitação de recursos dos computadores, essas linguagens foram fortemente influenciadas pelas linguagens de máquina e pela arquitetura de Von Neumman dos computadores. A eficiência computacional era o foco principal das LPs porque os recursos como memória e processadores eram escassos. FORTRAN e COBOL são exemplos de linguagens que surgiram nessa época.

Na medida que os recursos computacionais se desenvolviam, os computadores iam se tornando mais poderosos e úteis. Novamente, a atividade de programação se tornava um gargalo para a disseminação dos sistemas computacionais. No final dos anos 60, as LPs passaram a enfocar a eficiência na produtividade dos programadores. Surgiram as LPs que enfatizavam a programação estruturada. PASCAL e C são exemplos de linguagens que surgiram nessa época.

[1.2] ML não é considerada uma linguagem puramente funcional. Ela também possui características de uma linguagem imperativa.

Com o aumento da complexidade dos sistemas computacionais, uma nova técnica de programação passou a ser o foco das LPs no final dos anos 70 e início dos anos 80 - a abstração de dados. Essas LPs enfocavam a construção modularizada de programas e bibliotecas e o conceito de tipos abstratos de dados. MODULA-2 e ADA são exemplos de linguagens que surgiram nessa época.

Durante os anos 80 e 90 houve uma vasta disseminação do uso de computadores pessoais e das estações de trabalho. Surge a indústria do software e com ela a necessidade de se produzir e atualizar software rapidamente. O reuso passa a ser um conceito central para a produtividade no desenvolvimento de software. Para atender esse requisito são desenvolvidas as LPs orientadas a objetos. SMALLTALK, EIFFEL, C++ e JAVA são exemplos de linguagens que surgiram nessa época.

É importante dizer que muitas linguagens foram incorporando novas características na medida que se constatava a sua necessidade. Assim, versões atuais de FORTRAN e COBOL, por exemplo, já incorporam os conceitos de programação estruturada.

Cabe dizer ainda que as linguagens declarativas evoluíram em paralelo com as imperativas. LISP surgiu no final dos anos 50 e PROLOG no início dos anos 70. O maior interesse no desenvolvimento dessas linguagens tem sido demonstrado no meio acadêmico, em particular, nas áreas de pesquisa sobre Linguagens de Programação e Inteligência Artificial.

Apresenta-se a seguir uma breve descrição da origem e principais características de algumas das LPs mais conhecidas.

1.7.1 Origem de LPs

- **FORTRAN** (1957): Desenvolvida inicialmente por Backus para computadores IBM. Destinou-se a aplicações numérico-científicas (caracterizadas por poucos dados e muita computação). Enfatizava eficiência computacional (por exemplo, não havia alocação dinâmica de memória). Não enfocava eficiência dos programadores (por exemplo, as estruturas de controle eram todas baseadas no comando *GOTO*). Versões atuais de FORTRAN incorporaram avanços das outras LPs.

- **LISP** (1959): Criada por John McCarthy no MIT. Adota o paradigma funcional. Apropriada para processamento simbólico. Ainda hoje é a LP mais usada na Inteligência Artificial. COMMON LISP e SCHEME são dialetos.

- **ALGOL** (1960): Criada por um comitê de especialistas. Primeira LP com sintaxe formalmente definida. Importância teórica enorme, tendo influenciado todas as LPs imperativas subsequentes, embora ela própria não tenha sido muito usada (até hoje se usa o termo ALGOL–`like`).

- **COBOL** (1960): Criada por comitê de especialistas. Primeira LP encomendada pelo Departamento de Defesa Americano (DoD). Destinada para aplicações comerciais (caracterizada por muitos dados e pouca computação). Tentou enfatizar legibilidade (LP mais próxima do inglês), mas acabou comprometendo redigibilidade.

- **BASIC** (1964): Criada por Kemeny e Kurtz na Universidade de Darmouth. Objetivava ser de fácil aprendizado para uso por estudantes de artes e ciências humanas.

- **PASCAL** (1971): Criada por Niklaus Wirth. Foi projetada para ser usada no ensino de programação estruturada. Enfocou a simplicidade.

- **C** (1972): Criada por Dennis Ritchie no Bell Labs. Projetada para ser usada no desenvolvimento de sistemas de programação (em particular, para a implementação do sistema operacional UNIX).

- **PROLOG** (1972) - Criada por Comerauer e Roussel, na Universidade de Aix-Marseille, com o auxílio de Kowalski, da Universidade de Edinburgo. Adota o paradigma lógico, sendo bastante usada em Inteligência Artificial.

- **SMALLTALK** (1972): Criada por Alan Key e Adele Goldberg no Xerox PARC. Primeira LP totalmente orientada a objetos. O ambiente de programação de SMALLTALK introduziu o conceito de interfaces gráficas com o usuário que hoje é amplamente utilizado.

- **ADA** (1983): Criada pela empresa Cii-Honeywell Bull, liderada pelo francês Jean Ichbiah, vencedora de licitação promovida pelo DoD para atender à demanda de uma linguagem de programação de alto-nível padronizada. Demandou o maior esforço para o desenvolvimento de uma LP, envolvendo centenas de pessoas durante oito anos. LP muito grande e complexa. Apropriada para programação concorrente e sistemas de tempo real.

- **C++** (1985): Criada por Bjarne Stroustrup no Bell Labs. Projetada para ser uma extensão de C com orientação a objetos. Tinha como requisito não implicar em perda de eficiência em relação ao código em C. Responsável pela rápida aceitação da orientação a objetos. Se tornou uma LP muito complexa.

- **JAVA** (1995): Criada pela SUN para ser usada na construção de softwares para sistemas de controle embutido (tais como eletrodomésticos), mas acabou não sendo usada para este fim. Baseou-se fortemente em C++, mas é bem mais simples. É uma LP orientada a objetos. Não utiliza explicitamente o conceito de ponteiros e foi projetada para enfatizar a portabilidade. Tem se tornado amplamente utilizada por causa da sua confiabilidade e portabilidade, pelo advento da INTERNET e porque os programadores de C e C++ a aprendem facilmente.

1.8 Considerações Finais

Nesse capítulo foram apresentados diversos temas importantes para o entendimento dos conceitos discutidos no resto desse livro. Em particular, é importante ter compreendido como cada uma das propriedades apresentadas na seção 1.3 podem influenciar o projeto, implementação e uso das LPs. Ter uma boa noção sobre como LPs podem ser especificadas e implementadas também contribui para a compreensão de diversos tópicos subseqüentes.

Por fim, vale repetir que o foco desse livro será na discussão das LPs que se enquadram sobre o paradigma imperativo, isto é, linguagens que adotam o paradigma estruturado, orientado a objetos ou concorrente. Vale ressaltar também que os exemplos serão dados primordialmente nas linguagens C, C++ e JAVA.

1.9 Exercícios

1. Identifique problemas de legibilidade e redigibilidade nas LPs que conhece. Verifique se existem casos nos quais essas propriedades são conflitantes.

2. Identifique problemas de confiabilidade e eficiência nas LPs que conhece. Verifique se existem casos nos quais essas propriedades são conflitantes.

3. Identifique problemas de falta de ortogonalidade nas LPs que conhece. Esses problemas comprometem a facilidade de aprendizado da LP?

4. Reusabilidade e modificabilidade muitas vezes contribuem para a melhoria uma da outra. Dê exemplos de situações nas quais isso ocorre.

5. Identifique situações nas quais a busca por eficiência computacional compromete a portabilidade de LPs e vice-versa.

6. Uma LP sempre pode ser implementada usando tanto o método de compilação quanto o de interpretação? Em caso positivo, discuta se existem LPs que se ajustam melhor a um método de implementação do que a outro. Em caso negativo, apresente um exemplo de uma LP na qual só se pode utilizar um método de implementação e justifique.

7. Faça uma análise léxica, sintática e semântica das seguintes linhas de código C e descreva quais as conclusões obtidas:

```
int a, i;
int b = 2, c =3;
a = (b + c) * 2;
i = 1 && 2 + 3 | 4;
```

8. Enumere e explique quais os principais fatores que influenciaram a evolução das LPs imperativas.

9. Induzir a legibilidade, confiabilidade e reuso de programas são algumas das propriedades desejáveis em Linguagens de Programação. Mostre, através de exemplos (um para cada propriedade) retirados de linguagens de programação conhecidas, como elas podem cumprir estes papéis e justifique os seus exemplos.