1. INTRODUÇÃO

A primeira Linguagem de Programação (LP) de alto nível foi projetada durante a década de 50. Desde então, LP têm sido uma área de estudo fascinante e produtiva. Simplificadamente, uma LP pode ser definida como uma notação formal para descrever a execução de algoritmos em computador. Sendo assim, as LP não podem ser encaradas como entidades isoladas, pois elas são ferramentas utilizadas no desenvolvimento de *software* e com esse objetivo é que devem ser analisadas e estudadas. Por isso, é necessário entender a organização das LP, com ênfase na compreensão dos seus conceitos abstratos e dos elementos de sua representação durante o processo de execução [SIL 88, WAT 90].

Entretanto, para entender o relacionamento entre uma linguagem de programação e um método de projeto, também é importante compreender que as linguagens de programação podem forçar um certo estilo de programação, geralmente chamado "paradigma de programação". Por exemplo, como será visto nos próximos capítulos, *Smalltalk* é uma linguagem orientada a objetos. Ela força o desenvolvimento de programas baseados em classes de objetos como a unidade de modularização. Similarmente, *Fortran* e *Pascal* são linguagens procedurais, obrigando o desenvolvimento de programas baseados em rotinas como a unidade de modularização. Linguagens que forçam um paradigma de programação específico podem ser chamadas "orientadas a paradigma". Em geral não há necessidade de um relacionamento um para um entre paradigmas e linguagens de programação. Algumas linguagens, de fato, são "neutras" e suportam diferentes paradigmas. Por exemplo, *C++* suporta o desenvolvimento de programas procedurais e orientados a objetos.

Os paradigmas de linguagens de programação mais proeminentes, com ênfase no estilo de decomposição de programas que eles promovem, são:

- Programação Procedural: estilo de programação convencional onde os programas são decompostos em "passos" de processamento que executam operações complexas. Rotinas são usadas como unidades de modularização para definir tais "passos" de processamento. Exemplos de linguagens com este paradigma são *Pascal* e *C*.
- Programação Funcional: estilo de programação que tem origem na teoria das funções matemáticas. Enfatiza o processamento de valores através do uso de expressões e funções. As funções são os blocos de construção primários do programa; elas podem ser passadas livremente como parâmetro e podem ser construídas e retornadas como parâmetros resultantes de outras funções. Algumas LP que suportam este paradigma são *ML*, *Scheme* e *LISP*.
- Programação de Tipos Abstratos de Dados: reconhece tipos abstratos de dados como a unidade da modularidade do programa. *CLU* foi a primeira linguagem projetada para suportar este paradigma.
- Programação Orientada a Objetos: enfatiza a definição de classes de objetos. Instâncias de classes são criadas através do programa conforme a necessidade, durante a execução dos programas. Este estilo é baseado na definição de hierarquias de classes e seleções em tempo de execução de unidades para executar. *Smalltalk*, *Eiffel* e *Java* são linguagens representativas desta classe. *C++* e *Ada* também suportam este paradigma.
- Programação Declarativa: enfatiza a descrição declarativa de um problema, ao invés da decomposição do problema em uma implementação algorítmica. Tais programas são mais próximos de uma especificação do que a implementação tradicional. Linguagens lógicas, como *PROLOG*, e linguagens baseadas em regras, como *OPS5* e *CLIPS*, são representativas desta classe de linguagens [GHE 97].

Sendo assim, após uma introdução aos tópicos sobre classificação das LP e sobre seus elementos sintáticos e semânticos, feita nas próximas seções, alguns paradigmas de LP serão descritos. No capítulo 2 são apresentadas as características das linguagens imperativas (programação procedural). Na seqüência são introduzidos os conceitos do paradigma de orientação a objetos e suas propriedades. Os paradigmas concorrente e funcional são discutidos nos capítulos 4 e 5, respectivamente. Finalmente, o último capítulo aborda um estudo de caso que visa possibilitar a compreensão em detalhe das características das linguagens concorrentes e das linguagens funcionais de programação.

1.1. Motivação

As linguagens de programação fazem parte do "coração" da Ciência da Computação, pois correspondem às ferramentas que são utilizadas para comunicação, não apenas com os computadores, mas também entre as pessoas, e que são indispensáveis para programação do computador. O desafio de projetar características de uma LP que suportem uma declaração clara, a complexidade de colocar juntas diferentes características para construir uma linguagem útil, e o desafio de usar apropriadamente estas características para facilitar a construção dos algoritmos fazem parte do estudo de linguagens de programação [GHE 97].

Na verdade existem muitas razões para se estudar linguagens de programação, desde que se considere mais do que as características das linguagens e procure-se pesquisar os conceitos básicos de projeto de LP e seus efeitos na implementação da linguagem. Sendo assim, cinco razões principais podem ser identificadas rapidamente:

- Para melhorar o entendimento da linguagem que está sendo utilizada: muitas linguagens oferecem recursos que quando usados adequadamente trazem muitos benefícios ao programador, mas quando utilizados de maneira incorreta podem levar a um grande e desnecessário aumento do tempo de processamento ou à geração de erros lógicos [PRA 75].
- Para fazer um uso melhor da linguagem de programação: o estudo das estruturas das LP facilitará o entendimento da função e implementação de tais estruturas. Então, quando o usuário for programar, ele estará mais apto a usar a linguagem no grau máximo de sua funcionalidade de uma maneira eficiente. O entendimento do poder de uma linguagem permite que se utilize este poder. Em outras palavras, quando um programador busca dados e estruturas de programa adequados para a solução de um problema, existe a tendência dele pensar apenas em estruturas que são expressas nas linguagens com as quais ele está familiarizado. Porém, estudando-se as construções fornecidas por um grande número de linguagens, e a maneira na qual estas construções são implementadas, um programador aumenta o seu "vocabulário" de programação. O entendimento da implementação de tais construções também é muito importante, pois para usar uma determinada construção enquanto se está trabalhando com uma linguagem que não a forneça diretamente, é necessário que o programador a implemente utilizando os elementos disponíveis na linguagem [PRA 75, DER 90].
- Para permitir uma melhor escolha da linguagem de programação: a partir do conhecimento de uma variedade de linguagens de programação é possível escolher a linguagem certa para um projeto particular, reduzindo desta maneira a quantidade de código necessária.
- Para facilitar o aprendizado de uma nova linguagem: o conhecimento de uma variedade de construções de linguagens de programação e técnicas de implementação possibilita que o programador aprenda a trabalhar com uma nova linguagem de programação mais facilmente. Além disso, o programador rapidamente consegue ver com maior clareza como a linguagem é apropriadamente usada, quais construções são mais custosas para usar e quais são relativamente baratas [PRA 75].
- Para tornar mais fácil o projeto de uma nova linguagem: este benefício é mais importante do que parece inicialmente. Poucas pessoas quiseram ou tiveram a oportunidade de projetar suas próprias LP. Entretanto, considerando-se que a linguagem é um meio de comunicação entre uma pessoa e um computador, então cada sistema de computador que é desenvolvido deve ter uma linguagem incorporada para fornecer uma interação homem↔máquina. Um bom entendimento dos princípios da LP pode auxiliar no projeto desta interface. Além disso, muitas LP modernas possuem a propriedade de serem extensíveis de várias maneiras. Isto significa que o programador pode aperfeiçoar a linguagem através da adição de novos tipos de dados e operadores. Nestas linguagens, cada programa, na verdade, consiste no projeto de uma nova linguagem no sentido de que o programador tem o poder de aumentar a linguagem original [DER 90].

O estudo de LP é importante para qualquer programador, uma vez que este aprenderá a entender e utilizar mais eficientemente as facilidades encontradas nas LP atuais. Projetistas de LP, bem como projetistas de *hardware* e gerenciadores de *software*, também beneficiam-se deste estudo. Ao entender as necessidades de implementação de uma LP, projetistas de *hardware* podem aperfeiçoar a maneira na qual as máquinas suportam as linguagens. Já os gerenciadores poderão tomar diferentes decisões de acordo com as características da LP utilizada [MAC 87].

A partir do que foi descrito, deduz-se que há muito mais para se estudar em LP do que simplesmente compreender as características fornecidas por uma variedade de linguagens. Na verdade, muitas características semelhantes das linguagens enganam, uma vez que a mesma característica em duas linguagens diferentes pode ser implementada de duas maneiras diferentes, e, consequentemente, as duas versões podem diferir grandemente no custo de uso [PRA 75]. Assim, estudando-se paradigmas de linguagens, é possível se tornar um usuário inteligente de LP. Isto inclui a habilidade de: escolher linguagens apropriadas para diferentes aplicações, fazer uso efetivo e eficiente de uma linguagem no desenvolvimento de *software*, e aprender rapidamente novas linguagens [DER 90].

1.2. Conceito de Linguagem de Programação

O meio mais eficaz de comunicação entre pessoas é a linguagem (língua ou idioma), que consiste num conjunto de convenções e regras sistemático para comunicação de idéias ou troca de informações. Com uma linguagem natural, como português, esta comunicação ocorre entre pessoas e a linguagem é usada tanto na forma falada, como na forma escrita. Linguagens de programação, por sua vez, diferem das linguagens naturais de várias maneiras. Primeiro, a comunicação ocorre entre uma pessoa e uma máquina (computador). A segunda maior diferença é o conteúdo da comunicação, que, no caso da LP, é conhecido como programa. Programas são seqüências de instruções que descrevem as tarefas a serem realizadas para alcançar a solução de um determinado problema, e devem ser escritos numa LP para que possam ser executados num computador. Um terceiro exemplo de característica de comunicação através de uma LP, é o meio utilizado. Desde que o computador é o futuro receptor, isto significa que os programas são representados simbolicamente como conjuntos de caracteres, em oposição, por exemplo, aos sons.

Uma definição de trabalho para uma linguagem de programação é: “Uma linguagem de programação é uma linguagem com o objetivo de ser usada por uma pessoa para expressar um processo através do qual um computador pode resolver um problema” [DER 90]. Em outras palavras, a LP faz a ligação entre o pensamento humano (muitas vezes de natureza não estruturada) e a precisão requerida para o processamento pela máquina. Os quatro componentes chave nesta definição de linguagem de programação são:

- Computador: a máquina que executará o processo descrito através do programa;
- Pessoa: o programador que serve como a origem da comunicação;
- Processo: a atividade que está sendo descrita através do programa;
- Problema: o sistema atual ou ambiente onde o problema surgiu.

Outra definição usual é: Linguagem de programação é um conjunto de termos e de regras que permitem a formulação de instruções a um computador. Desta forma, a LP é composta por dois elementos: Vocabulário, que consiste num conjunto de símbolos, e Gramática, que é o conjunto de regras para usar o vocabulário.

A descrição de uma linguagem é geralmente dividida em duas partes, descritas detalhadamente na seção 1.5, que são:

- Sintaxe: é o conjunto de regras que determina quais construções são corretas para formação dos programas e quais não são; em outras palavras, preocupa-se com a “forma” dos programas (como expressões, comandos, declarações, etc. são colocados juntos para formar programas).
- Semântica: é a descrição da maneira que um programa sintaticamente correto é interpretado ou executado; em outras palavras, preocupa-se com o “significado” dos programas (como o programa vai se comportar quando executado no computador) [DER 90, WAT 90].

Torna-se interessante comentar que o principal objetivo de uma LP é dar suporte ao programador no desenvolvimento dos sistemas. Isto inclui assistência no projeto, implementação, teste, verificação e manutenção do *software*. Existe uma série de características numa linguagem que contribuem para este objetivo. Entre as características gerais que definem uma boa linguagem pode-se destacar:

- Simplicidade: clareza e concisão semântica (linguagem com um mínimo número de conceitos e estruturas), e clareza sintática (sintaxe deve representar cada conceito de uma maneira apenas).

- Suporte para abstração de dados: representação de um objeto deve incluir somente os atributos relevantes.
- Expressividade: refere-se a facilidade com que um objeto pode ser representado; a linguagem deve oferecer estruturas de dados e de controle apropriadas.
- Ortogonalidade: refere-se a interação entre conceitos, isto é, o grau com que diferentes conceitos podem ser combinados uns com os outros de uma maneira consistente; ortogonalidade reduz o número de exceções das regras de uma linguagem, tornando mais fácil o seu aprendizado e memorização.
- Suporte à manutenção e portabilidade: habilidade de manter programas que devem ser fáceis de entender e alterar; é afetada pelas características anteriores.
- Eficiência: a avaliação precisa de uma linguagem, baseada em critérios pré-definidos, é extremamente importante; as medidas mais comuns são a eficiência da execução do programa, da tradução do programa, e da criação, teste e uso do programa [DER 90, PRA 75, SIL 88].

1.3. Classes e Gerações

É possível afirmar que um programa de computador é uma abstração da realidade, onde abstração (seção 1.5.2) consiste no processo de identificar as qualidades ou propriedades importantes do fenômeno que está sendo modelado, e ignorar todas as propriedades irrelevantes. Levando em consideração o nível de abstração, torna-se possível identificar três classes de linguagens de programação, descritas a seguir e ilustradas na figura 1.1.

- Linguagem de Máquina: é uma linguagem usualmente baseada num código binário, específico para cada tipo de computador (ou microprocessador). Como a codificação das instruções é feita na forma final de execução (endereços e *opcode* em binário), esta linguagem pode ser usada diretamente pela máquina
- Linguagem de Baixo Nível: aqui começa a ser introduzida a abstração, pois os códigos binários são substituídos por mnemônicos. Neste caso é necessário usar um montador (*assembler*), que consiste num programa que lê o programa em baixo nível e converte os códigos mnemônicos para *opcodes*. O *assembly*, como é chamada a linguagem de baixo nível, é característico para cada equipamento, podendo no entanto ser manipulado com certa facilidade pelos programadores.
- Linguagem de Alto Nível: é a LP mais próxima da linguagem do homem, não requer conhecimento da arquitetura da máquina e é portável, isto é, independente da máquina. Neste caso, o programa ou código fonte precisa ser traduzido para linguagem de máquina para poder ser executado. O código ou programa objeto é o resultado da tradução quando é utilizado um compilador ou interpretador (seção 1.7).

Para entender melhor as LP também é interessante estudar um pouco da sua história. Dershem e Jipping [DER 90], então, estruturaram o histórico das LP em três períodos. O primeiro período que durou aproximadamente uma década, iniciou em 1955 e abrangeu o desenvolvimento das linguagens de programação de primeira geração. O segundo período, de 1965 a 1971, foi um tempo de consolidação em torno do modelo de uma linguagem, ALGOL 60, com o desenvolvimento de um grande número de linguagens derivadas dela, mas com novas extensões através da adição de novas características importantes. No período final, 1972 em diante, os resultados das pesquisas preliminares foram colocados juntos para introduzir novos modelos e abordagens para linguagens de programação.

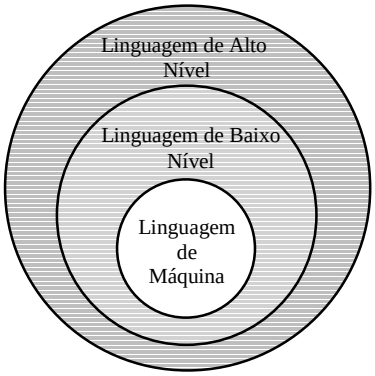

Figura 1.1 - Classes de linguagens de programação

Entretanto, cronologicamente, as LP são usualmente classificadas em cinco gerações, que serão descritas a partir de agora. A estrutura das linguagens de **primeira geração** são baseadas na estrutura dos computadores do início da década de 60. Isto é notório especialmente nas estruturas de controle, que correspondem às instruções de máquina. As instruções condicionais não são aninhadas e dependem fortemente do "*goto*" para construir qualquer estrutura de controle. Uma exceção é o comando de iteração definida, como por exemplo o laço "*do*" do *Fortran*, que é hierárquico nesta primeira geração. Procedimentos recursivos não são permitidos na maioria das linguagens, *Basic* é uma exceção, e há geralmente só um modo de passagem de parâmetro, tipicamente por referência.

Também pode-se observar que esta geração de LP é orientada para máquina nos tipos de estrutura de dados fornecidos. As estruturas de dados primitivas encontradas são números de ponto flutuante e fixo de várias precisões, caracteres e valores lógicos. Os construtores de estruturas de dados são vetores e registros. Assim como nas estruturas de controle, há pouca facilidade para organização de dados hierárquica, isto é, as estruturas de dados não podem ser aninhadas, sendo a estrutura de registro do *Cobol* uma exceção. Nomes de variáveis, por sua vez, são amarradas diretamente e estaticamente às localizações de memória, pois não há gerenciamento dinâmico de memória. As estruturas sintáticas são caracterizadas através de comandos lineares, orientados a cartão. Além disso, a maioria destas linguagens de primeira geração possui *labels* numéricos que sugerem endereços de máquina. As convenções léxicas usuais, são ignorar espaços em branco e reconhecer palavras chave no contexto. Resumindo, as características marcantes são a orientação para máquina e as estruturas lineares.

A primeira linguagem da **segunda geração** foi o *Algol-60*, e suas características são típicas de toda geração. Inicialmente pode-se dizer que as estruturas desta geração constituem aperfeiçoamentos e generalizações das equivalentes na primeira geração. As estruturas de dados são muito próximas da primeira geração, tendo algumas generalizações, tais como vetores dinâmicos, porém continuam lineares. As linguagens de segunda geração geralmente não possibilitam tipos definidos pelo usuário, e uma das suas grandes contribuições são as estruturas de nome, que são hierarquicamente aninhadas. Isto permite um melhor controle do espaço do nome e uma alocação de memória dinâmica e eficiente. Outra característica é a introdução das estruturas de bloco e de controle estruturado, que eliminam a necessidade do uso de "*goto*". Várias estruturas de controle também foram aperfeiçoadas, permitindo o uso de procedimentos recursivos e a opção de escolha do modo de passagem de parâmetro. Nesta geração surgiu a distinção entre variáveis locais e globais e nas suas estruturas sintáticas houve o avanço em direção às linguagens livre de formatos, com convenções léxicas independentes de máquina. Construções caras e não muito eficientes também proliferaram na segunda geração. Um exemplo de linguagem é *PL/I*.

A **terceira geração** é caracterizada por compensar os "excessos" da segunda geração, enfatizando a simplicidade e a eficiência, e é simbolizada pelo *Pascal*. As estruturas de dados mostram a troca da ênfase da máquina para a aplicação. Isto pode ser observado através do fornecimento de tipos de dados definidos pelo usuário, que permitem que programadores criem os tipos de dados necessários para as suas aplicações, e através dos construtores de tipos orientados à aplicação, tais como conjuntos e enumerações. A terceira geração também é caracterizada por permitir que estruturas de dados sejam aninhadas, isto é, organizadas hierarquicamente. As estruturas de nome constituem uma simplificação do Algol-60, e as linguagens desta geração geralmente possuem novas amarrações e construtores de tipos compostos, tais como conjuntos. Estruturas de controle são simplificadas, como pode-se observar no laço "*for*" do *Pascal*, e criadas com ênfase nas aplicações, como por exemplo o "*case*" do *Pascal*.

Algumas características da **quarta geração** constituem apenas uma consolidação e correção das características da terceira geração. Entretanto, novas facilidades importantes, tal como o suporte para programação concorrente, também são fornecidas. A contribuição mais importante desta geração está no domínio das estruturas de nome, cuja principal característica é o fornecimento do encapsulamento, que suporta a separação da especificação e da definição e ocultar informações (abstração de dados). A maioria das linguagens de quarta geração permitem que módulos encapsulados sejam genéricos (ou usem polimorfismo). Outra contribuição significativa está nas estruturas de controle, desde que há o suporte a programação concorrente. Várias linguagens usam alguma forma de troca de mensagens como meio de sincronização e comunicação entre tarefas concorrentes, e possuem um mecanismo de tratamento de exceções para gerenciar erros do sistema e do usuário. Por outro lado, o *framework* básico destas linguagens ainda é seqüencial. Os construtores de estruturas de dados são similares à

terceira geração, porém alguns problemas, tal como a passagem de parâmetros de vetores, foram corrigidos. As estruturas de dados primitivas tendem a ser mais complicadas do que antes, devido ao controle e precisão de tipos numéricos. Finalmente, as estruturas sintáticas da quarta geração são mais abrangentes, tendo como exceção a preferência por estruturas completamente agrupadas entre parênteses. Exemplos de linguagens desta geração são *Ada*, *Modula-2*, e *Clu*.

Na **quinta geração**, marcada pela execução de vários experimentos, não foi identificada uma idéia predominante de programação. Entretanto, três paradigmas em especial são identificados: linguagens funcionais, lógicas e orientadas a objetos. As linguagens funcionais enfatizam o uso de funções puras e a redução de operações de atribuição. Isto leva ao uso da recursão como método de iteração e da notação polonesa como estrutura sintática básica. Sua estrutura de dados principal é a lista, e as estruturas de controle básicas são as expressões condicionais e a recursão. Estas linguagens possuem um alto nível de abstração, permitem a avaliação dos programas em várias ordens diferentes, o que as tornam adequadas para uso em computadores paralelos, e são adequadas para aplicações matemáticas. Um exemplo deste tipo de linguagem é *LISP*. As linguagens lógicas, por sua vez, são de alto nível e orientadas a aplicação. A programação com este tipo de linguagem é clara, rápida, precisa e simples, pois os programas são descritos em termos de predicados e há uma separação da lógica e do controle durante a programação. *Prolog* é um exemplo de linguagem lógica. Já a programação orientada a objetos fornece um modelo de programação diferente, onde os objetos possuem propriedades semelhantes aos objetos do mundo real. Linguagens orientadas a objetos concentram-se diretamente no comportamento dos objetos no tempo. Um exemplo deste tipo de linguagem é o *Smalltalk* [MAC 87].

1.4. Escolha de uma Linguagem de Programação

Devido a existência de um grande número de LP o programador deve escolher qual é a mais adequada para ele usar em cada aplicação. Existem várias considerações que devem ser levadas em conta, técnicas e não técnicas, estratégicas e táticas. Resumidamente, pode-se dizer que a escolha da linguagem mais adequada está intimamente ligada a três fatores: complexidade do sistema a ser desenvolvido; características peculiares da aplicação (por exemplo, sistemas de tempo real); e facilidades que as linguagens oferecem ao suporte de metodologias de desenvolvimento [SIL 88, WAT 90].

Para pequenos programas, feitos e mantidos por uma pessoa, é provável que a linguagem mais adequada seja aquela melhor dominada pela pessoa. No entanto, para grandes sistemas com programação em tempo real, vários tipos de exceções a serem tratadas e muitas pessoas envolvidas, a escolha da linguagem mais adequada deve ser feita criteriosamente [SIL 88]. Assim, Dershem e Jipping [DER 90] apresentaram sete critérios importantes que devem considerados quando se está decidindo qual LP utilizar. Estes critérios são:

- Implementação: Refere-se ao tradutor da linguagem que é utilizado. Neste caso, existem duas considerações importantes relacionadas à implementação, sua disponibilidade e sua eficiência. A disponibilidade tem impacto na decisão de quando usar a LP dada uma aplicação e a plataforma onde ele deverá ser utilizada. A eficiência refere-se à velocidade de execução de programas objetos criados pelo tradutor. Por exemplo, *Fortran* é muito utilizado porque o seu compilador contém várias características de otimização e produz um código objeto extremamente eficiente.
- Conhecimento do Programador: Apesar de ser interessante assumir que todos os programadores estão igualmente aptos a programar em qualquer linguagem, na prática não funciona bem assim. Primeiramente, como resultado da educação e experiência do programador, ele/ela tem uma maior facilidade para uma ou duas linguagens e é mais eficiente quando usa estas linguagens. Apesar de que aprender a usar uma nova e mais apropriada linguagem para uma determinada aplicação é uma ótima experiência, o empregador raramente está disposto a dar o suporte financeiro, principalmente porque também envolve o treinamento de vários programadores que irão fazer a validação, teste e manutenção dos programas. Para exemplificar, é por isso que, apesar da disponibilidade de muitas linguagens mais eficientes, ainda encontram-se vários programas desenvolvidos em *Cobol*.
- Portabilidade: se a possibilidade de executar uma aplicação em uma grande variedade de computadores é importante, então a portabilidade é um critério significativo na escolha da linguagem. Linguagens que aderem a padrões, como *Fortran*, *Cobol* e *Ada*, são escolhas mais seguras em situações onde há linguagens cujas

implementações são dependentes da máquina e resultam em mais tempo e custo quando portadas para um novo sistema.

- Sintaxe: algumas aplicações adaptam-se melhor à sintaxe de uma linguagem do que as outras. Por exemplo, a sintaxe de *Fortran* foi projetada para atender às exigências de programas matemáticos, enquanto linguagens mais novas como *Pascal* e *Ada* possuem uma sintaxe que facilita a declaração de estruturas de controle.
- Semântica: pode ser um fator significativo na escolha da linguagem dada uma aplicação específica, pois se uma aplicação requer ou é facilitada por uma certa característica da linguagem, então a linguagem que fornece esta(s) característica(s) deve ser escolhida. Por exemplo, se a programação concorrente é necessária, deve-se escolher uma LP concorrente, como *Ada*.
- Ambiente de Programação: a disponibilidade de um rico ambiente de programação para suportar o desenvolvimento de sistemas pode influenciar na escolha da linguagem. Se a linguagem está disponível em um ambiente que fornece um bom editor, um *debugger* simbólico, um sistema de controle de código fonte, janelas, ou outras ferramentas de desenvolvimento, o esforço necessário para produzir sistemas pode ser significativamente reduzido. Apesar de algumas destas ferramentas estarem disponíveis em ambientes independentes da linguagem, tal como o sistema operacional, freqüentemente elas são específicas à implementação de uma linguagem. A disponibilidade de bibliotecas também é um fator que deve ser considerado.
- Modelo de Processamento: Uma consideração final na seleção de uma LP é o modelo de processamento no qual a linguagem é baseada. For exemplo, se uma aplicação requer uma quantidade significativa de busca heurística, uma linguagem que segue o modelo orientado a lógica seria apropriada. Simulações, por sua vez, são geralmente mais facilmente implementadas usando uma LP orientada a objetos.

Watt [WAT 90] também fez um levantamento das questões que devem ser levadas em consideração na escolha da LP, dividindo-as em questões estratégicas e táticas. As questões estratégicas, que envolvem projeto e integração de *software*, são:

- Abstração: o quanto a linguagem dá suporte ao conceito de abstração, que permite separar o que um módulo deve fazer de como ele será implementado;
- Desenvolvimento de grandes programas: se a linguagem permite que programas sejam construídos a partir de módulos escritos e verificados separadamente;
- Reutilização de *software*: se a linguagem permite a reutilização de código, e o quanto isto pode acelerar o desenvolvimento de novos projetos.

As questões táticas, que referem-se a detalhes de programação na linguagem proposta, são:

- Modelagem: se a linguagem fornece tipos e operações associadas que são adequadas para modelagem de objetos na área de aplicação do projeto;
- Nível: o quanto uma linguagem é alto ou baixo nível, encorajando o programador a pensar em termos relacionados à área de aplicação, ou a pensar em detalhes de código;
- Segurança: se a linguagem foi projetada de tal maneira que facilita a detecção de erros de programação, preferencialmente em tempo de compilação;
- Eficiência: se a linguagem é capaz de ser implementada eficientemente;
- Compilador: se existe um compilador de boa qualidade disponível para a linguagem;
- Familiaridade: o quanto o programador está familiarizado com a linguagem, ou se tem disponível um treinamento adequado.

É interessante comentar que nenhuma destas questões são específicas para uma área de aplicação particular. Isto indica que qualquer linguagem razoável que passa por estas questões pode ser bem aceita na maioria das áreas de aplicações. Isto também explica porque Pascal é tão satisfatório (e insatisfatório) em uma grande variedade de aplicações [WAT 90].