Universidade Do Vale do Itajaí

Campus Kobrasol

Paradigmas de Programação

Resumo – Livro Conceito de Linguages de Programação, Seções 1.1, 1.3 e 1.5

Aluno: Maurício Macário de Farias Junior

Professora: Fernanda dos Santos Cunha

São José, 2018, 20 de Fevereiro

Razões para estudar conceitos de linguagens de programação

É natural que os estudantes se perguntem como se beneficiarão com o estudo de conceitos de linguagens de programação. Afinal, muitos outros tópicos em ciência da computação são merecedores de um estudo sério. A seguir, temos uma lista de potenciais vantagens de estudar esses conceitos:

- *Capacidade aumentada para expressar ideias:* Acredita-se que a profundidade com a qual as pessoas podem pensar é influenciada pelo poder de expressividade da linguagem que elas usam para comunicar seus pensamentos, portanto pessoas com apenas um fraco entendimento da linguagem natural são limitadas na complexidade de seus pensamentos, particularmente na abstração.

  Programadores apresentam a mesma limitação. As linguagens usadas impoem restrições nos tipos de estruturas de controle ou de dados e abstrações que eles podem usar. Conhecer uma variadade de recursos maior pode reduzir essas limitações, os programadores podem aumentar a faixa de seus processos mentais.

  Mesmo que o programador seja forçado a usar uma linguagem, as construções de outras linguagens podem ser simuladas em outras que não as oferecem suporte diretamente. Por exemplo, um programador C que aprendeu a estrutura e os usos de matrizes associativas em Perl pode projetar estruturas que as simulem em C, por isso o estudo de conceitos de linguagens de programação constrói uma apreciação de recursos e construções valiosas das linguagens e incentiva os programadores a usa-las.
- *Embasamento para escolher linguagens adequadas:* Programadores que não tiveram muita educação formal em Ciência da Computacação tipicamente sabem uma ou duas linguagens diretamente relevante para os projetos da empresa atual, e também muitos programadores receberam sua educação formal a muito tempo e as linguagens vistas não são mais utilizadas, isso faz com que seja escolhida sempre a mesma linguagem para todos projetos, mesmo que ela conclua tal projeto de forma ineficaz, se conhecessem uma variedade maior de opções isso poderia ser evitado.

  É sempre melhor usar um recurso cujo projeto tenha sido integrado em uma linguagem do que usar uma simulação normalmente de manipulação mais difícil e menos segura em uma linguagem que não suporta tal recurso.
- *Habilidade aumentada para aprender novas linguagens:* As linguagens de programação ainda estão em evolução, isso exige um aprendizado contínuo do profissional, o processo de aprendizado de uma linguagem pode ser longo e difícil, especialmente para alguém que esteja confortável com apenas uma ou duas e nunca examinou os conceitos de linguagens de programação de modo geral, caso o tenha adquirido, facilita o aprendizado.
- *Melhor entendimento da importância da implementação:* Em alguns casos, um entendimento de questões de implementação leva a por que as linguagens foram projetadas de uma determinada forma. Esse conhecimento muitas vezes leva à habilidade de usar uma linguagem de maneira mais inteligente e como ela foi projetada para ser usada.

  Certos tipos de erros em programas podem ser encontrados e corrigidos apenas por programadores que conhecem alguns detalhes de implementação relacionados. Outro

benefício é permitir a visualização do funcionamento da execução das contruções de uma linguagem

- *Melhor uso de linguagens já conhecidas:* Ao estudar os conceitos de linguagens de programação, os programadores podem aprender sobre partes antes desconhecidas e não utilizadas das linguagens que eles já trabalham e começar a utilizá-las.
- *Avanço geral da computação:* Em alguns casos, pode-se concluir que uma linguagem se tornou amplamente usada, ao menos em parte, porque aqueles em posições de escolha não estavam suficientemente familiarizafdos com conceitos de linguagens de programação.

Critérios de Avaliações de Linguagens

É necessário um conjunto de critérios de avaliação para analisar o impacto dos recursos no processo de desenvolvimento de software incluindo a manutenção. Note que alguns dos critérios são amplos – e, de certa forma, vagos – como a facilidade de escrita, enquanto outros são construções específicas de linguagens, como o tratamento de exceções. Apesar de a discussão parecer implicar que os critérios têm uma importância idêntica, não é o caso.

	CRITÉRIOS		
Característica	**LEGIBILIDADE**	**FACILIDADE DE ESCRITA**	**CONFIABILIDADE**
Simplicidade	•	•	•
Ortogonalidade	•	•	•
Tipos de dados	•	•	•
Projeto de sintaxe	•	•	•
Suporte para abstração		•	•
Expressividade		•	•
Verificação de tipos			•
Tratamento de exceções			•
Apelidos restritos			•

- **Legibilidade**

Um dos critérios mais importantes para julgar uma linguagem de programação é a facilidade com a qual os programas podem ser lidos e entendidos. Até os anos 1970 o desenvolvimento de software era amplamente pensado em termos de escrita de código, após isso, o conceito de ciclo de vida de software (Booch, 1987) foi desenvolvido; a codificação foi relegada a um papel muito menor, e a manutenção foi reconhecida como uma parte principal do ciclo, particularmente em termos de custo. Como a facilidade de manutenção é determinada, em grande parte, pela legibilidade dos programas, a

legibilidade se tornou uma medida importante da qualidade dos programas e das linguagens de programação, o que representou um marco na sua evolução. A legibilidade deve ser considerada no contexto do domínio do problema.

- Simplicidade Geral: A simplicidade geral de uma linguagem de programação afeta fortemente sua legibilidade. Uma linguagem com muitas construções básicas é mais difícil de aprender do que uma com poucas. Os programadores que precisam usar uma linguagem grande aprendem um subconjunto dessa linguagem e ignoram outros recursos, isso gera problemas caso o leitor estiver familiarizado com outro subconjunto.

Outra característica de uma linguagem de programação que pode ser um complicador é a multiplicidade de recursos – ou seja, haver mais de uma maneira de realizar uma operação.

Um terceiro problema em potencial é a sobrecarga de operadores, na qual um operador tem mais de um significado. Apesar de ser útil, pode levar a uma redução da legibilidade se for permitido aos usuários criar suas próprias sobrecargas e eles não o fizerem de maneira sensata. Por exemplo, é aceitável sobrecarregar o operador + para usá-lo tanto para a adição de inteiros quanto para a adição de valores de ponto flutuante.

A simplicidade em linguagens pode, é claro, ser levada muito ao extremo. Por exemplo, a forma e o significado da maioria das sentenças de uma linguagem assembly são modelos de simplicidade, o extremo disso torna os programas escritos em assembly menos legíveis.

- Ortogonalidade: em uma linguagem de programação significa que um conjunto relativamente pequeno de construções primitivas pode ser combinado a um número relativamente pequeno de formas para construir as estruturas de controle e de dados da linguagem. Além disso, cada possível combinação de primitivas é legal e significativa. Por exemplo, considere os tipos de dados. Suponha que uma linguagem tenha quatro tipos primitivos de dados (inteiro, ponto flutuante, ponto flutuante de dupla precisão e caractere) e dois operadores de tipo (vetor e ponteiro). Se os dois operadores de tipo puderem ser aplicados a eles mesmos e aos quatro tipos de dados primitivos, um grande número de estruturas de dados pode ser definido.

A ortogonalidade vem de uma simetria de relacionamentos entre primitivas. Uma falta de ortogonalidade leva a exceções às regras de linguagem. Por exemplo, deve ser possível, em uma linguagem de programação que possibilita o uso de ponteiros, definir que um aponte para qualquer tipo específico definido na linguagem. Entretanto, se não for permitido aos ponteiros apontar para vetores, muitas estruturas de dados potencialmente úteis definidas pelos usuários não poderiam ser definidas.

A ortogonalidade é fortemente relacionada à simplicidade: quanto mais ortogonal o projeto de uma linguagem, menor é o número necessário de exceções às regras da linguagem. Menos exceções significam um maior grau de regularidade no projeto, o que torna a linguagem mais fácil de aprender, ler e entender.

Como um exemplo da dependência do contexto, considere a seguinte expressão em C “(a + b)” Ela significa que os valores de a e b são obtidos e adicionados juntos. Entretanto, se a for um ponteiro, afeta o valor de b. Por exemplo, se a aponta para um valor de ponto flutuante que ocupa quatro bytes, o valor de b deve ser ampliado – nesse caso,

multiplicado por 4 – antes que seja adicionado a a. Logo, o tipo de a afeta o tratamento do valor de b. O contexto de b afeta seu significado.

Muita ortogonalidade também pode causar problemas. Talvez a linguagem de programação mais ortogonal seja o ALGOL 68 (van Wijngaarden et al.,1969). Cada construção de linguagem no ALGOL 68 tem um tipo, e não existem restrições nesses tipos. Além disso, mais construções produzem valores. Essa liberdade de combinações permite construções extremamente complexas. Por exemplo, uma sentença condicional pode aparecer no lado esquerdo de uma atribuição, com declarações e outras sentenças diversas, desde que o resultado seja um endereço. Essa forma extrema de ortogonalidade leva a uma complexidade desnecessária.

Simplicidade em uma linguagem é, ao menos em parte, o resultado de uma combinação de um número relativamente pequeno de construções primitivas e um uso limitado do conceito de ortogonalidade.

ombinação de simplicidade e ortogonalidade. Uma linguagem funcional, como LISP, é uma na qual as computações são feitas basicamente pela aplicação de funções para parâmetros informados. Em contraste a isso, em linguagens imperativas como C, C++ e Java, as computações são normalmente especificadas com variáveis e sentenças de atribuição. As linguagens funcionais oferecem a maior simplicidade de um modo geral, porque podem realizar tudo com uma construção, a chamada à função, a qual pode ser combinada com outras funções de maneiras simples. Essa elegância simples é a razão pela qual alguns pesquisadores de linguagens são atraídos pelas linguagens funcionais como principal alternativa às complexas linguagens não funcionais como C++. Outros fatores, como a eficiência, entretanto, têm prevenido que as linguagens funcionais sejam mais utilizadas.

- Tipos de dados: A presença de mecanismos adequados para definir tipos e estruturas de dados é outro auxílio significativo à legibilidade. Por exemplo, suponha que um tipo numérico seja usado como uma flag porque não existe nenhum tipo booleano na linguagem. Em tal linguagem, poderíamos ter uma atribuição como: timeOut = 1 O significado dessa sentença não é claro. Em uma linguagem que inclui tipos booleanos, teríamos: timeOut = true O significado dessa sentença é perfeitamente claro.

- Projeto da Sintaxe*:* A sintaxe, ou forma, dos elementos de uma linguagem tem um efeito significativo na legibilidade dos programas. A seguir, estão dois exemplos de escolhas de projeto sintáticas que afetam a legibilidade:

* *Formato dos identificadores*: Restringir os identificadores a tamanhos muito curtos piora a legibilidade.

* *Palavras especiais*: A aparência de um programa e sua legibilidade são fortemente influenciadas pela forma das palavras especiais de uma linguagem (por exemplo, while, class e for). O método para formar sentenças compostas, ou grupos de sentenças, é especialmente importante, principalmente em construções de controle. Algumas linguagens têm usado pares casados de palavras especiais ou de símbolos para formar grupos. C e seus descendentes usam chaves para especificar sentenças compostas. Todas essas linguagens sofrem porque os grupos de sentenças são sempre terminados da mesma forma, o que torna difícil determinar qual grupo está sendo finalizado quando um end ou um } aparece. Outra questão importante é se as palavras especiais de uma linguagem

podem ser usadas como nomes de variáveis de programas. Se puderem, os programas resultantes podem ser bastante confusos.

* *Forma e significado:* Projetar sentenças de maneira que sua aparência ao menos indique parcialmente seu propósito é uma ajuda óbvia à legibilidade. A semântica, ou significado, deve advir diretamente da sintaxe, ou da forma. Em alguns casos, esse princípio é violado por duas construções de uma mesma linguagem idênticas ou similares na aparência, mas com significados diferentes, dependendo talvez do contexto. Em C, por exemplo, o significado da palavra reservada static depende do contexto no qual ela aparece.

Uma das principais reclamações sobre os comandos de shell do UNIX (Raymond, 2004) é que a sua aparência nem sempre sugere sua funcionalidade.

• Facilidade de escrita

A facilidade de escrita é a medida do quão facilmente uma linguagem pode ser usada para criar programas para um domínio. A maioria das características de linguagem que afetam a legibilidade também afeta a facilidade de escrita. Isso é derivado do fato de que o processo de escrita de um programa requer que o programador releia sua parte já escrita.

Como ocorre com a legibilidade, a facilidade de escrita deve ser considerada no contexto do domínio de problema alvo de uma linguagem.

- Simplicidade e ortogonalidade: Se uma linguagem tem um grande número de construções, alguns programadores não estarão familiarizados com todas. Essa situação pode levar ao uso incorreto de alguns recursos e a uma utilização escassa de outros que podem ser mais elegantes ou mais eficientes (ou ambos) do que os usados. Por outro lado, muita ortogonalidade pode prejudicar a facilidade de escrita. Erros em programas podem passar despercebidos quando praticamente quaisquer combinações de primitivas são legais.

- Suporte à abstração: As linguagens de programação podem oferecer suporte a duas categorias de abstrações: processos e dados. Um exemplo simples da abstração de processos é o uso de um subprograma para implementar um algoritmo de ordenação necessário diversas vezes em um programa. Como um exemplo de abstração de dados, considere uma árvore binária que armazena dados inteiros em seus nós. Tal árvore poderia ser implementada em uma linguagem que não oferece suporte a ponteiros e gerenciamento de memória dinâmica usando um monte (heap), como no Fortran 77, com o uso de três vetores inteiros paralelos, onde dois dos inteiros são usados como índices para especificar nós filhos. Em C++ e Java, essas árvores podem ser implementadas utilizando uma abstração de um nó de árvore na forma de uma simples classe com dois ponteiros (ou referências) e um inteiro.

- Expressividade: A expressividade em uma linguagem pode se referir a diversas características. Em uma linguagem como APL (Gilman e Rose, 1976), expressividade significa a existência de operadores muito poderosos que permitem muitas computações com um programa muito pequeno. Em geral, uma linguagem expressiva especifica computações de uma forma conveniente, em vez de deselegante. Por exemplo, em C, a notação count++ é mais conveniente e menor do que count = count + 1. Além disso, o

operador booleano and then em Ada é uma maneira conveniente de especificar avaliação em curto-circuito de uma expressão booleana. A inclusão da sentença for em Java torna a escrita de laços de contagem mais fácil do que com o uso do while, também possível. Todas essas construções aumentam a facilidade de escrita de uma linguagem.

- ## Confiabilidade

Um programa é dito confiável quando está de acordo com suas especificações em todas as condições. As seguintes subseções descrevem diversos recursos de linguagens que têm um efeito significativo na confiabilidade dos programas em uma linguagem.

- Verificação de tipos: A verificação de tipos é a execução de testes para detectar erros de tipos em um programa, tanto por parte do compilador quanto durante a execução de um programa. A verificação de tipos é um fator importante na confiabilidade de uma linguagem. Como a verificação de tipos em tempo de execução é cara, a verificação em tempo de compilação é mais desejável.

- Tratamento de exceções: A habilidade de um programa de interceptar erros em tempo de execução (além de outras condições não usuais detectáveis pelo programa), tomar medidas corretivas e então continuar é uma ajuda óbvia para a confiabilidade.

- Utilização de apelidos: Em uma definição bastante informal, apelidos são permitidos quando é possível ter um ou mais nomes para acessar a mesma célula de memória. Atualmente, é amplamente aceito que o uso de apelidos é um recurso perigoso em uma linguagem de programação. A maioria das linguagens permite algum tipo de apelido – por exemplo, dois ponteiros configurados para apontarem para a mesma variável.

- Legibilidade e facilidade de escrita: Tanto a legibilidade quanto a facilidade de escrita influenciam a confiabilidade. Quanto mais fácil é escrever um programa, mais provavelmente ele estará correto. A legibilidade afeta a confiabilidade tanto nas fases de escrita quanto nas de manutenção do ciclo de vida. Programas difíceis de ler são também difíceis de escrever e modificar.

- ## Custo

O custo total definitivo de uma linguagem de programação é uma função de muitas de suas características.

Primeiro, existe o custo de treinar programadores para usar a linguagem, que é uma função da simplicidade, da ortogonalidade da linguagem e da experiência dos programadores.

Segundo, o custo de escrever programas na linguagem. Essa é uma função da facilidade de escrita da linguagem, a qual depende da proximidade com o propósito da aplicação em particular.

Terceiro, o custo de compilar programas na linguagem. Um grande impeditivo para os primeiros usos de Ada era o custo proibitivamente alto da execução dos compiladores da primeira geração.

Quarto, o custo de executar programas escritos em uma linguagem é amplamente influenciado pelo projeto dela. Uma linguagem que requer muitas verificações de tipos em

tempo de execução proibirá uma execução rá- pida de código, independentemente da qualidade do compilador. Apesar de eficiência de execução ser a principal preocupação no projeto das primeiras linguagens, atualmente é considerada menos importante.

Otimização é o nome dado à coleção de técnicas que os compiladores podem usar para diminuir o tamanho e/ou aumentar a velocidade do código que produzem.

O quinto fator é o custo do sistema de implementação da linguagem. Um dos fatores que explica a rápida aceitação de Java são os sistemas de compilação/interpretação gratuitos que estavam disponíveis logo após seu projeto ter sido disponibilizado ao público.

O sexto fator é o custo de uma confiabilidade baixa. Se um aplicativo de software falha em um sistema crítico, como uma usina nuclear ou uma máquina de raio X para uso médico, o custo pode ser muito alto.

A consideração final é o custo de manter programas, que inclui tanto as correções quanto as modificações para adicionar novas funcionalidades. O custo da manutenção de software depende de um número de características de linguagem, principalmente da legibilidade. A importância da facilidade de manutenção de software não pode ser subestimada. De todos os fatores que contribuem para os custos de uma linguagem, três são os mais importantes: desenvolvimento de programas, manutenção e confiabilidade mas é claro, outros critérios podem ser usados para avaliar linguagens de programação. Um exemplo é a portabilidade, a facilidade com a qual os programas podem ser movidos de uma implementação para outra. A generalidade (a aplicabilidade a uma ampla faixa de aplicações) e o fato de uma linguagem ser bem definida (em relação à completude e à precisão do documento oficial que define a linguagem) são outros dois critérios.

Uma nota final sobre critérios de avaliação: os critérios de projeto de linguagem têm diferentes pesos quando vistos de diferentes perspectivas. Implementadores de linguagens estão preocupados principalmente com a dificuldade de implementar as construções e recursos da linguagem. Os usuários estão preocupados primeiramente com a facilidade de escrita e depois com a legibilidade. Os projetistas são propensos a enfatizar a elegância e a habilidade de atrair um grande número de usuários. Essas características geralmente entram em conflito.

- ***Categorias de linguagens***

Linguagens de programação são normalmente divididas em quatro categorias: imperativas, funcionais, lógicas e orientadas a objetos. Descrevemos como as linguagens mais populares que suportam a orientação a objetos cresceram a partir de linguagens imperativas.

As linguagens visuais são uma subcategoria das imperativas, fornecem uma maneira simples de gerar interfaces gráficas de usuário para os programas. Por exemplo, em VB.NET, o código para produzir uma tela com um controle de formulário, como um botão ou uma caixa de texto, pode ser criado com uma simples tecla. Tais capacidades estão agora disponíveis em todas as linguagens .NET

Alguns autores se referem às linguagens de scripting como uma categoria separada de linguagens de programação. Entretanto, linguagens nessa categoria são mais unidas entre si

por seu método de implementação, interpretação parcial ou completa, do que por um projeto de linguagem comum. As linguagens de scripting, dentre elas Perl, JavaScript e Ruby, são imperativas em todos os sentidos.

Uma linguagem de programação lógica é um exemplo de uma baseada em regras. Em uma linguagem imperativa, um algoritmo é especificado em muitos detalhes, e a ordem de execução específica das instruções ou sentenças deve ser incluída. Em uma linguagem baseada em regras, entretanto, estas são especificadas sem uma ordem em particular, e o sistema de implementação da linguagem deve escolher uma ordem na qual elas são usadas para produzir os resultados desejados. Essa abordagem para o desenvolvimento de software é radicalmente diferente daquelas usadas nas outras três categorias de linguagens e requer um tipo completamente diferente de linguagem.

Nos últimos anos, surgiu uma nova categoria: as linguagens de marcação/linguagens de programação híbridas. As de marcação não são de programação.

Diversas linguagens de propósito especial têm aparecido nos últimos 50 anos. Elas vão desde a Report Program Generator (RPG), usada para produzir relatórios de negócios; até a Automatically Programmed Tools (APT), para instruir ferramentas de máquina programáveis; e a General Purpose Simulation System (GPSS), para sistemas de simulação.