RESUMO

Livro: Linguagens de Programação - Princípios e Paradigmas, 2ª Ed, 2009, de Tucker e Noonan

**LINGUAGENS IMPERATIVAS**

* A programação imperativa é o paradigma de programação mais antigo e bem desenvolvido. Ela surgiu com os primeiros computadores, na década de 1940, e seus elementos espelham diretamente as características arquiteturais dos computadores modernos também.
* todas as linguagens imperativas incluem a atribuição como um elemento central. Além disso, elas suportam declarações de variáveis, expressões, comandos condicionais, laços e abstração procedural.
* Comandos de atribuição forneciam a capacidade de se atualizar dinamicamente o valor armazenado em um local de memória, enquanto os comandos condicionais e de ramificação podiam ser combinados para permitir que um conjunto de comandos fosse pulado ou executado repetidamente. atribuição, sequenciamento de comandos baseados em memórias, condições e comandos de ramificação são “**completas quanto a Turing**”.
* Uma linguagem de programação imperativa é, assim, a que é “**completa quanto a Turing**” e também suporta determinadas características comuns que surgiram com a evolução do paradigma de programação imperativa:

• Estruturas de controle.

• Entrada/saída.

• Manipulação de exceções e erros.

• Abstração procedural.

• Expressões e atribuição.

• Suporte de biblioteca para estruturas de dados.

* A característica de ser “**completa quanto a Turing**” é importante porque fornece uma medida pela qual um conjunto mínimo de recursos de uma linguagem pode expressar todos os algoritmos que possam ser concebidos. Essa característica não é exclusiva das linguagens imperativas – linguagens funcionais, lógicas e orientadas a objeto também são completas quanto a Turing, no sentido de que qualquer uma dessas linguagens é igualmente capaz de expressar qualquer algoritmo.
* os programas são modelados como “algoritmos mais estruturas de dados”.
* O processo de refinamento gradual utiliza abstração procedural desenvolvendo um algoritmo da sua forma mais geral para uma implementação específica.
* A **semântica de uma atribuição** é simples. Na ausência de erros, a expressão é avaliada para um valor, que é então copiado para o destino, ou seja, a maioria das linguagens imperativas usa semântica de cópia. Expressões são escritas com o uso de operadores aritméticos e lógicos familiares, assim como as chamadas ocasionais a funções-padrão fornecidas pela linguagem.
* O processo de abstração procedural permite ao programador se preocupar principalmente com a interface entre a função e o que ela calcula, ignorando os detalhes de como o cálculo é executado.

**LINGUAGENS FUNCIONAIS**

* A programação funcional emergiu como um paradigma distinto no início da década de 1960. Sua criação foi motivada pela necessidade dos pesquisadores no desenvolvimento de inteligência artificial e em seus subcampos – computação simbólica, prova de teoremas, sistemas baseados em regras e processamento de linguagem natural.
* Linguagens de programação funcional denominadas **puras** eliminam a noção de célula de memória de uma variável em favor da noção matemática; isto é, uma variável dá nome a uma expressão imutável, que também elimina o operador de atribuição.
* Uma linguagem funcional é **pura** se não houver nenhum conceito de um operador de atribuição ou de uma célula de memória; caso contrário, dizemos que ela é **impura**. No entanto, muitas linguagens de programação funcionais retêm alguma forma de operador de atribuição e são, portanto, **impuras**.
* Uma função tem **transparência referencial** se o seu valor depende somente dos valores de seus argumentos.
* A **base da programação funcional** é o cálculo lambda, desenvolvido por Church (1941);
* Uma linguagem de programação funcional é essencialmente um cálculo lambda aplicado com valores constantes e funções embutidas.
* Na expressão lambda (lx · M), dizemos que a variável x está ligada à subexpressão M. Uma variável ligada é uma variável cujo nome é igual ao nome do parâmetro; caso contrário, dizemos que a variável é livre (**free**).
* Uma alternativa para a estratégia da **avaliação rápida** é chamada **avaliação lenta**, na qual um argumento para uma função não é avaliado (ela é adiada) até que ele seja necessário.
* Uma **vantagem da avaliação rápida** é a eficiência, em que cada argumento passado a uma função é avaliado apenas uma vez, enquanto na **avaliação lenta** um argumento para uma função é reavaliado cada vez que ele é usado, e isso pode ocorrer mais de uma vez. Uma **vantagem da avaliação lenta** é que ela permite certas funções interessantes a serem definidas, de modo que não podem ser implementadas em linguagens rápidas.
* Um aspecto importante da programação funcional é que as funções são tratadas como valores de primeira classe. Um nome de uma função pode ser passado como um parâmetro, e uma função pode retornar outra função como valor. Uma função dessas às vezes é chamada **forma funcional**. Um exemplo de uma **forma funcional** seria uma função *g* que toma como parâmetro uma função e uma lista (ou uma sequencia de valores) e aplica a função dada a cada elemento na lista, retornando uma lista.
* **Avaliação preguiçosa (lenta)** – técnica usada para atrasar a avaliação de uma expressão até que o seu valor seja necessário e que evita avaliações repetidas. Benefícios: capacidade de definir estruturas controle como funções regulares (melhor que usar primitivas internas); a capacidade de definir estruturas de dados potencialmente infinitas (implementação mais simples de alguns algoritmos) e aumento de desempenho (evitando cálculos desnecessários e condições de erro na avaliação de expressões compostas).

**LINGUAGENS LÓGICAS**

* Um programa lógico expressa as especificações para soluções de problemas com o uso de expressões em lógica matemática. Esse estilo evoluiu a partir das necessidades dos pesquisadores em processamento de linguagem natural e na prova automática de teorema.
* A cláusula de Horn é uma variante particular de lógica predicativa que está por trás da sintaxe da Prolog. Definição: Uma cláusula de Horn tem uma parte mais importante h, que é um atributo, e um corpo, que é uma lista de atributos p1 , p2 , . . ., pn .
* Características

– Programas são relações entre E/S

– As características da solução são especificadas, mas o processo completo de obtê-la não o é

– Estilo declarativo, como no paradigma funcional

– Foco no o que fazer (declarativo), e não no como fazer (imperativo)

* Cláusulas de Horn

– 4 classificações:

• Incondicional

• Positiva

• Condicional

• Negativa

**LINGUAGEM CONCORRENTE**

* Um programa concorrente é um programa projetado para ter dois ou mais contextos de execução. Chamamos um programa desses de **multithread**, porque mais de um contexto de execução pode estar ativo simultaneamente.
* Objetivos da programação concorrente:

1. Reduzir o tempo total de processamento (múltiplos processadores)
2. Aumentar confiabilidade e disponibilidade (ִprocessadores distribuídos)
3. Obter especialização de serviços (sistemas operacionais, simuladores)
4. Implementar aplicações distribuídas (ִcorreio eletrônico)

* Um programa paralelo é um programa concorrente no qual vários contextos de execução, ou threads, estão ativos simultaneamente. Para nossa finalidade, não há diferença entre um programa concorrente e um programa paralelo.
* Um programa distribuído é um programa concorrente projetado para ser executado simultaneamente em uma rede de processadores autônomos que não compartilham a memória principal, com cada thread rodando em seu próprio processador separado. Em um sistema operacional distribuído, por exemplo, o mesmo programa (como um editor de texto) pode ser executado por vários processadores, com cada instância tendo seu próprio contexto de execução separado dos outros (ou seja, sua própria janela). Isso não é o mesmo que um programa multithread, no qual os dados podem ser compartilhados entre diferentes contextos de execução.
* Uma seção de código que requer acesso exclusivo a uma variável compartilhada é chamada de **seção crítica de uma thread.**
* Programas concorrentes requerem comunicação ou interação interthreads. A comunicação ocorre pelas seguintes razões:

1. Uma thread, às vezes, requer acesso exclusivo a um recurso compartilhado, como uma fi la de impressão, por exemplo, uma janela de um terminal ou um registro em um arquivo de dados.
2. Uma thread, às vezes, precisa trocar dados com outra thread. Em ambos os casos, as duas threads em comunicação devem sincronizar sua execução para evitar confl ito ao adquirir recursos ou para fazer contato ao trocar dados. Uma thread pode se comunicar com outras threads por meio de:
3. Variáveis compartilhadas: esse é o mecanismo primário usado em Java, e também pode ser usado em Ada.
4. Passagem de mensagens: esse é o mecanismo primário usado em Ada: técnica que resolve o problema de o que fazer quando múltiplos pedidos simultâneos são feitos por outras tarefas para se comunicar com uma em especial. Usa forma de não-determinismo para assegurar justiça na escolha de qual pedido será atendido primeiro.
5. Parâmetros: são usados em Ada em conjunto com a passagem de mensagens.

Threads normalmente cooperam umas com as outras para resolver um problema. No entanto, é altamente desejável manter a comunicação entre threads em um nível mínimo; isso torna o código mais fácil de entender e deixa cada thread rodar em sua própria velocidade sem ser retardada por protocolos complexos de comunicação.

* **Concorrência física**:

1. diversas unidades do mesmo programa literalmente executam simultaneamente, supondo que mais de um processador esteja disponível.

* **Concorrência lógica**:

1. programador e programa supõem a existência de múltiplos processadores fornecendo concorrência real quando, de fato, a execução real dos programas está se dando intercaladamente em um único processador.

* **Problema do produtor-consumidor**: originário do desenvolvimento de SOs, nos quais uma unidade de programa produz algum valor de dados ou recurso e outra unidade o usa. Os dados são colocados em um retentor (buffer) de armazenamento pela unidade produtora e removido dela pela consumidora, de forma sincronizada. Não se permite remoção no buffer vazio, nem colocação no buffer cheio.
* **Semáforo** – estrutura de dados que consiste em um número inteiro e uma fila que armazena descritores de tarefas (armazena todas as informações relevantes sobre o estado de execução de uma tarefa).
* **Monitor** – técnica para sincronizar 2 ou mais tarefas q compartilham um recurso em comum (dispositivo de hardware ou região da memória). Assim, o programador não precisa ter acesso às primitivas para tal, tendo que realizar o bloqueio e desbloqueio de recursos manualmente.