Universidade Federal de São João del-Rei

Departamento do Curso de Ciência da Computação
Programa de Graduação

Relatório sobre Paradigmas de Programação

Aluno: Jardel Felipe de Carvalho
Professor orientador: Dárlinton B. Feres Carvalho

Setembro
2018

Universidade Federal de São João del Rei
Departamento do Curso de Ciência da Computação
Programa de Graduação

Relatório sobre Paradigmas de Programação

Relatório do Primeiro Trabalho Prático da Disciplina de Conceitos de Linguagens de Programação do Curso de Ciência da Computação da Universidade Federal de São João del Rei.

Aluno: Jardel Felipe de Carvalho

Professor orientador: Dárlinton B. Feres Carvalho

Setembro
2018

Conteúdo

1 Introdução

1.1 Contexto

O ser humano e a sua inventividade vem revolucionando o campo das linguagens de programção desde o primeiro computador da história.
Na busca por maneiras menos abstratas de se realizar tarefas que necessitam da programação de computadores se deu o surgimento de diversos paradigmas de programação, esses paradigmas podem afetar a forma de pensar do programador e consequentemente a forma com que o problema é transcrito do mundo real para a máquina.
Atualmente existe um número muito grande de linguagens de programação, cada uma delas com suas respectivas peculiaridades, então para organização foi criado um conjunto de categorias que auxiliam na classificação de acordo com o paradigma de cada linguagem.

1.2 Descrição do trabalho

Este trabalho consiste na seleção de três linguagens com diferentes paradigmas de programação para o desenvolvimento de variadas maneiras de se calcular o n-ésimo termo da série de Fibonacci.

2 Materiais e métodos

2.1 Paradigmas utilizados

Para o desenvolvimento dos algorítmos foram selecionados três paradigmas de programação, sendo eles os paradigmas funcional, imperativo e orientado a objetos. O paradigma funcional é baseado na definição do cálculo Lambda apresentado por Alonzo Church no ano de 1930. O paradigma imperativo é fortemente baseado na arquitetura de Von Neumann que tem como característica marcante a troca de informações entre memória principal, memória secundária, centro de processamento, unidade de controle e mecanismos de entrada e saída, proposto por John von Neumann em 1945. Por fim, o paradigma orientado a objetos teve sua primeira aparição com a linguagem de programação Simula 67 no ano de 1967, a característica marcante deste paradigma é o alto nível de abstração e o encapsulamento dos comportamentos e atributos de cada entidade presente no programa.

2.2 Linguagens utilizadas

Para o paradigma funcional foi utilizada a linguagem de programação Haskell que teve a primeira aparição em 1990. Haskell é uma linguagem fortemente tipada, de execução híbrida e totalmente recursiva. Traz consigo uma facilidade para desenvolver programas expressívos de maneira rápida, também proporciona uma execução de forma eficiente graças a um recurso chamado transparência referêncial que garante com que os resultados gerados com maior frequência sejam guardados, permitindo com que os mesmos sejam consultados quando preciso provendo assim uma alta taxa de otimização.A linguagem Haskell também conta com um recurso chamado avaliação preguiçosa, que é caracterizada pela computação de funções somente quando é necessário, ao contrario da linguagem C que por sua vez tende a fazer otimizações em tempo de compilação que quebram o conceito de avaliação preguiçosa.
A linguagem C foi utilizada no desenvolvimento imperativo, criada por Dennis Ritchie em 1972 na Bell Labs com o proposito de se desenvolver o sistema operacional Unix.
Para o paradigma orientado a objetos foi utilizada a linguagem de programação Java, desenvolvida por programadores da Sun Microsystems no ano de 1995, é atualmente considerada uma das linguagens mais utilizadas no mercado e seu aspecto marcante é a portabilidade devido a utilização de uma máquina virtual Java (JVM) para interpretação dos bytecodes gerados após a compilação

2.3 Estratégias de solução

As variações de implementação foram baseadas em três formatos de algoritmos diferentes para cálculo do n-ésimo termo da sequência de Fibonacci, o imperativo, o recursivo e o recursivo em cauda.

2.3.1 Sequencial

Conta com complexidade O(n) da qual n é o termo da sequência. Este tipo de função geralmente é mais segura pois garante que o consumo de memória seja restrito ao número de variáveis alocadas dinâmicamente ou estaticamente por uma única chamada da função. Esse formato de algorítmo é menos sugestivo e de complicada compreensão se comparado aos outros formatos de implementação. Vale pontuar o esforço que deve ser empregado para se extrair o comportamento de todas as instruções a cada iteração, portanto é notável a baixa expressividade e legibilidade presente nesta representação.

```
fib (n):
    r <- 0
    s <- 1
    Para N = n até N = 1 faça:
        t <- s
        s <- s + r
        r <- t
    Fim Para
    termo <- s
```

2.3.2 Recursão comum

Ordem de complexidade O(σ^n) onde σ é o número áureo e n é o termo da sequência. É a forma mais custosa computacionalmente, porém a que traz maior elegância e expressividade. Outro detalhe importante a se relatar é a grande semelhança com a fórmula proposta também por Leonardo Fibonacci.

```
fib (n):
    Se n = 0 então:
        Retorne 0
    Senão Se n = 1 então:
        Retorne 1
    Senão:
        Retorne fib (n - 1) + fib (n - 2)
    Fim Se
```

2.3.3 Recursão em Cauda

Apesar de ser recursivo, a complexidade é semelhante ao sequencial, ou seja O(n) em que n é o termo da sequência. Recursões em cauda tem como caracterísca principal a última operação realizada sendo a própria chamada recursiva da função. São mais eficientes que as recursões comuns e também consomem menos memória, pois não necessitam tanto do espaço de armazenamento da pilha assim como a versão sequêncial. Não é tão expressiva e legível quanto a versão recursiva comum, porém, pode-se deduzir que sua compreensão é mais facilitada do que na versão sequêncial.

```
fib (n, r, s):
    Se n = 1 então:
        Retorne s
    Senão:
        Retorne calcfib (n - 1, s, r + s)
    Fim Se

calcfib (n):
    p <- 1
    termo <- calcfib (n, 0, p)
```

3 Resultados e discussões

3.1 Implementação

3.1.1 Haskell

Como Haskell é uma linguagem completamente recursiva os programas desenvolvidos nesta linguagem foram baseadas nas versões recursiva comum e em cauda do algorítmo utilizado para cálculo do termo da sequência (pseudocódigos citados anteriormente).

3.1.2 Haskell recursão comum

Na versão comum foram criados três formatos do algorítmo. O primeiro conforme é mostrado abaixo aparece de forma bastante expressiva e legível, é de fácil identificação os elementos formais da equação recursiva $F_n = F_{n-1} + F_{n-2}$, definida por Leonardo Fibonacci.

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Os passos base fib 0 e fib 1 mostram claramente os dois valores necessários para se calcular o segundo termo da sequência.
O segundo formato aparece com um conjunto de estruturas condicionais.

```
fib :: Int -> Int
fib n = if n == 0 then 0
        else if n == 1 then 1
        else fib (n - 1) + fib (n - 2)
```

Repare a verbosidade presente nesta implementação, apesar de não ser tão expressivo quanto a versão anterior, cumpre o quesito leitura com comandos de se então.
Por fim, para a terceira implementação em recursão comum temos o seguinte código.

```
fib :: Int -> Int
fib n
    |n == 0 = 0
    |n == 1 = 1
    |otherwise = fib (n - 1) + fib (n - 2)
```

É a versão menos expressiva e legível das três, devido ao fato de utilizar recursos que só são de fácil compreensão para quem já possui contato com a linguagem. Cada símbolo pipe presente nesse programa é uma estrutura condicional que retorna resultados somente se o valor lógico for True para o teste após o pipe. A palavra chave otherwise (caso contrário) traz um pouco mais de expressividade e facilidade de escrita ao código assim como o uso dos pipes, porém não são recursos muito sugestivos se comparados a expressividade da primeira versão.

3.1.3 Haskell recursão em cauda

Para recursão em cauda tem-se a presença de duas funções conforme é mostrado a seguir.

```
fib :: Int -> Int
fib n = calcfib n 0 1

calcfib :: Int -> Int -> Int -> Int
calcfib 0 r _ = r
calcfib n r s = calcfib (n - 1) s (r + s)
```

Está versão possui também grande similaridade a versão recursiva em cauda presente no pseudocódigo acima citado. A função fib é chamada tendo como parâmetro o n-ésimo termo a ser calculado e após isso e feita uma chamada aninhada da função que calculará o valor do termo da sequência. Nota-se a presença de um símbolo underline no passo base de calcfib, este elemento signifca não importa, ou seja, quando a última chamada recursiva for realizada, n terá o valor zero e o resultado estará pronto para ser retornado a partir do elemento r. Nestas condições, não se faz necessário saber qual é o valor do elemento s, portanto, não importa o valor de s, assim o uso do underline se faz viável.

3.1.4 C

Com a linguagem C foi possível implementar as três versões do algorítmo de Fibonacci, sendo elas a sequêncial, recursiva e recursiva em cauda.

3.1.5 C sequêncial

Repare que são implementações de conteúdo mais denso.

```
long long int fib(int n) {
    long long int r = 0, s = 1, t, termo;
    for(int i = n; i >= 1; i--) {
        t = s;
        s = s + r;
        r = t;
    }
    termo = s;
    return termo;
}
```

Como a linguagem C é totalmente compilada, não existe o auxílio de um interpretador que realiza a alocação dos dados como no Haskell, com isso devemos explicitar quais são os tipos primitivos utilizados assim como também devemos fazer a alocação das variáveis. Esta função e de nível baixíssimo de expressividade, assim como é de difícil escrita e leitura. Em linguagens que permitem o uso da imperatividade é muito comum o acontecimento de efeitos colaterais devido ao seu baixo nível de abstração que induzem o programador a produzir falhas no programa ao contrário do Haskell que é totalmente imune a efeitos colaterais. A linguagem C possui três maneiras de se criar estruturas de repetição, no código anterior temos a utilização de um laço for. Abaixo podemos ver o mesmo código com um laço while.

```
long long int fib(int n) {
    long long int r = 0, s = 1, t, termo;
    while(n >= 1) {
        t = s;
        s = s + r;
        r = t;
        n--;
    }
    termo = s;
    return termo;
}
```

A segundo diferença mais notável nesta implementação e a necessidade de decrementar o valor de n dentro do escopo do laço while, isso se deve ao fato de não existir como no for um espaço para que esta operação seja feita. Isso pode afetar a legibilidade e a escrita do programa. Inferir onde decrementar o valor de uma variável de contagem é um processo que custa tempo e está sucetível a erros. O laço for é útil neste aspecto, pois após ser definida a operação de decremento dentro da linha declarativa do laço for, todas as outras operações desta estrutura de repetição serão pensadas em função da operação de decremento. A expresividade no uso de while ou for é equivalente.

Abaixo pode-se perceber a mesma implementação com a utilização do recurso goto.

```
long long int fib(int n) {
    long long int r = 0, s = 1, t, termo;
    loop:
        t = s;
        s = s + r;
        r = t;
        if(--n >= 1)
            goto loop;
        termo = s;
        return termo;
    }
}
```

Esta implementação possui uma certa similaridade a linguagem de programação Assembly. Em termos mais complicados, é como se o registrador PC (Program Counter) recebesse o endereço da label chamada loop todas as vezes que o valor n for maior ou igual a 1. O resultado final é semelhante ao de qualquer operação, entretanto é um código de compreensão dificultosa que se assemelha a uma linguagem de programação que caiu em desuso devido a baixa legibilidade e dificuldade de escrita. A expressividade nesta implementação também continua a mesma.

3.1.6 C recursão comum

Esta implementação é altamente expressiva porém carrega alguns problemas, sua escrita e leitura são dificultadas além do fato de ser implementado com base em um modelo de algorítmo com complexidade exponencial.

```
long long int fib(int n) {
    if(n == 0 || n == 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

3.1.7 C recursão em cauda

Esta implementação possui expressivadade um pouco reduzida e escrita mais dificultosa que a recursão comum, porém é mais otimizada e possui facilidade de leitura e escrita maiores se comparada a versão sequencial.

```
long long int calcfib
    (int n, long long int r, long long int s) {

    if(n == 1) return atual;
    return calcfib(n - 1, s, s + r);
}

long long int fib(int n) {
    return calcfib(n, 0, 1);
}
```

3.1.8 Java

Com a linguagem Java também foi possível implementar as três versões do algorítmo de Fibonacci, porém existem alguns detalhes com relação a variáveis de classe e pilha do método, aos modificadores de acesso para os atributos da classe e aos métodos extras utilizados.

3.1.9 Java sequêncial

Repare que as varáveis r, s, t e termo não precisam ser declaradas dentro do método pois já foram declaras no escopo da classe fib.

```
public class Fib {
    private long r, s, t, termo;
    public long fib(int n) {
        r = 0;
        s = 1;
        for(int i = n; i >= 1; i--) {
            t = s;
            s = s + r;
            r = t;
        }
        termo = s;
        return termo;
    }
}
```

A implementação acima traz maior legibilidade porém é mais inseguro e sucetível a efeitos colaterais, visto que as variáveis utilizadas continuam existindo mesmo após a execução do método. As variáveis são privadas, isso garante melhora a confiabilidade e a capacidade de manutenção.

Outra implementação do modelo sequêncial faz o uso de variáveis pertencentes ao escopo do método além do uso de um método privado que realiza a soma de dois termos da sequência numérica.

```
public class Fib {
    private long soma(long a, long b) {
        return a + b;
    }
    public long fib(int n) {
        long r, s, t, termo;
        r = 0;
        s = 1;
        for(int i = n; i >= 1; i--) {
            t = s;
            s = soma(s, r);
            r = t;
        }
```

```
            termo = s;
            return termo;
        }
    }
```

Este formato garante uma maior legibilidade e consequentemente maior manutenabilidade além do fato de garantir uma maior facilidade de escrita graças ao modificador de acesso private que contribui com o raciocínio do desenvolvedor no processo de criação permitindo com que o este faça o desenvolvimento de todo o programa a partir de pequenos módulos contendo algumas instruções.
A última implementação faz a utilização de um método estático para cálculo do termo, basicamente, não há nenhuma vantagem sobre a primeira implementação sequencial em Java, entretanto, a utilização da palavra chave static no método fib pode gerar efeitos colaterais em chamadas assíncronas deste método, visto que ele só existe na classe. Segue abaixo o exemplo.

```
public class Fib {
    public static long fib(int n) {
        ...
    }
}
```

3.1.10 Java recursivo

Implementação recursiva em Java. Apesar de ser em outro paradigma, carrega alta semelhança a implementação recursiva em C. A vantagem adicional é o encapsulamento que melhora a legibilidade e a escrita.

```
public class Fib {
    public long fib(int n) {
       if(n == 0 || n == 1) {
           return n;
       }
       return fib(n - 1) + fib(n - 2);
    }
}
```

3.1.11 Java recursivo em cauda

Implementação recursiva em cauda na linguagem Java agrega grande nível de semelhança a implementação em C. A vantagem adicional é o encapsulamento que melhora a legibilidade e a escrita.

```
public class Fib {
    public long fib(int n) {
        return calcfib(n, 0, 1);
    }
    private long calcfib(int n, long r, long s) {
        if(n == 1) {
            return s;
        }
        return calcfib(n - 1, s, s + r);
    }
}
```

3.2 Tempo de execução

Para os testes destinados a extração do tempo de execução para cada tipo de implementação nos três paradigmas de programação foram dadas as entradas 10, 20, 30, 40 e 50, onde cada um desses valores representa o n-ésimo termo da sequência de Fibonacci a ser calculado. Do somatório dos tempos de execução em uma implementação específica com diversas entradas foi calculada a sua média de execução. A fórmula a seguir representa este cálculo, $T_k = \frac{\sum_{k=1}^{5} t(fib(10k))}{5}$.
Observe atentamente nas seções a seguir a disparidade de tempo de execução com chamadas recursivas entre o Haskell e as demais linguagens, sendo elas C e Java.

3.2.1 Tempo Haskell

Implementação	Média (milisegundos)
Recurs.1	0.006 ms
Recurs.2	0.002 ms
Recurs.3	0.012 ms
Recurs. Cauda	0.002 ms

3.2.2 Tempo C

Implementação	Média (milisegundos)
Seq.1	0.000001 ms
Seq.2	0.000002 ms
Seq.3	0.000002 ms
Recursivo	25.16142 ms
Recurs. Cauda	0.000001 ms

3.2.3 Tempo Java

Implementação	Média (milisegundos)
Seq.1	0.00216 ms
Seq.2	0.00316 ms
Seq.3	0.00116 ms
Recusivo	3322.43980 ms
Recurs. Cauda	0.003280 ms

4 Análise dos Resultados

A partir dos critérios de avaliação para as linguagens de programação podemos concluir que a linguagem Haskell se sobressaiu no quesito expressividade e ausência de efeitos colaterais em contexto com C e Java.
Através das medições de tempo de execução e isolando os casos em que as diferenças de tempo são insignificantes, podemos perceber uma grande diferença de desempenho na execução de funções recursivas comuns entre as três linguagens.

Paradigma	Linguagem	Méd. Recursão (milisegundos)
Funcional	Haskell	0.012 ms
Imperativo	C	25.161 ms
O. Objetos	Java	3322.439 ms

Note que foi selecionado o pior tempo de execução em Haskell para comparação. Vale relembrar que a linguagem Haskell é totalmente recursiva.
Alterando a perspectiva, podemos perceber que a linguagem C foi a mais eficiente em linhas gerais.

Paradigma	Linguagem	Melhores tempos (milisegundos)
Funcional	Haskell	0.002 ms
Imperativo	C	0.000001 ms
O. Objetos	Java	0.00116 ms

5 Conclusão

Para os paradigmas Funcional, imperativo e orientado a objetos foi feita a seleção das linguagens de programação Haskell, C e Java. De acordo com os resultados foi possível perceber o ganho em desempenho da linguagem Haskell para cálculo do n-ésimo termo da sequência de Fibonacci de maneira recursiva. Java foi a linguagem que teve o pior tempo de execução neste quesito e C foi a linguagem que conseguiu a posição intermediária, contudo, é precipitado concluir que Haskell é em todas as situações a linguagem que garante a maior velocidade de execução, visto que a linguagem C conseguiu de forma sequêncial uma velocidade de execução muito maior do que Haskell em seu melhor tempo. Portanto, o que é possível concluir é que Haskell se sobresai no cálculo recursivo do n-ésimo termo da sequência de Fibonacci, isto em comparação com as linguagens de progração C e Java.
Apesar desta conclusão, é sensato avaliar que estes resultados aparecem no contexto de três linguagens e não três paradigmas, visto que o campo das linguagens de programação está sempre se renovando o que implica que pode ser possível que uma linguagem de programação orientada a objetos venha um dia a se tornar mais efiente que Haskell se for efetuado o mesmo teste feito neste relatório.
Para um conclusão mais precisa, seria necessário a avaliação das linguagens Haskell, C e Java em suas abordagens particulares para funções recursivas. A comparação de eficiência entre a transparência referêncial do Haskell para com a compilação de recursões pelo GCC assim como a interpretação de procedimentos recursivos pela JVM. Somente dessa forma seria possível aumentar a precisão dos resultados assim como o número de conclusões.

Bibliografia

SEBESTA, R. W. Conceitos de linguagens de programação, Aspectos preliminares. Porto Alegre, Brasil, Bookman. 2011.

Anexo