Universidade Federal São João del Rei
Curso de Ciência da Computação

Trabalho Prático 1 - Parte 2
Relatório

Conceitos de Linguagens de Programação

Aluno: Julio Cesar da Silva Rodrigues

Docente: Dárlinton Barbosa Feres Carvalho

Outubro
2022

Conteúdo

1 Introdução

Nesta seção, será apresentada uma breve introdução sobre a proposta do trabalho prático, e o que será abordado mais adiante neste relatório.

1.1 Contexto

Os primeiros registros de linguagens de programação surgiram na década de 50 com a *Fortran*. Desde então, com o avanço na área da tecnologia da informação, fez-se necessária a criação de novas linguagens com diversos propósitos de aplicação, assim como a criação de novos paradigmas de programação. Novas formas de pensar foram surgindo, de desenvolver o raciocínio na proposição de soluções para problemas, tudo isto disposto nas mais distintas formas observadas no código fonte, cada uma com suas peculiaridades em relação à vantagens e desvantagens.

1.2 Descrição do Trabalho

Neste trabalho, a proposta central é apresentar de forma breve e objetiva, as particularidades envolvidas na utilização de paradigmas de linguagens de programação distintos. Serão apresentadas as soluções desenvolvidas, e também uma análise superficial sobre vantagens e desvantagens encontradas não só no desenvolvimento de tais soluções, mas em aspectos gerais sobre cada paradigma de programação.

2 Materiais e Métodos

Nesta seção, serão apresentadas de forma breve as linguagens utilizadas no desenvolvimento de uma solução para o problema, assim como as estratégias aplicadas em cada uma, de acordo com os paradigmas de programação particulares à cada uma delas.

2.1 Linguagens

Como sugerido na especificação deste trabalho prático, ao todo foram utilizadas três linguagens de programação distintas: *Python*, *Prolog* e *Haskell*. Estas possuem como base, três paradigmas de programação diferentes: paradigma imperativo (embora *Python* seja uma linguagem multiparadigma, esta foi utilizada de forma imperativa no desenvolvimento da solução para este problema), paradigma lógico e paradigma funcional, respectivamente.

Python é uma linguagem de programação de propósito geral, ou seja, pode ser utilizada em diversos tipos de aplicações. Possui abstrações de altíssimo nível e versatilidade com sua sintaxe moderna e objetiva que produz códigos bastante enxutos.

Prolog é uma linguagem declarativa, onde geralmente são fornecidas descrições dos problemas à serem resolvidos, utilzando como recursos base, fatos e regras. Possui como fundamentos básicos a lógica de 1ª ordem, o que a torna mais voltada ao conhecimento e sua representação. Com o intuito de obter informação, são realizadas consultas (*queries*) que são computadas por um motor de inferência lógica.

Haskell é uma linguagem puramente funcional de propósito geral, que fornece abstrações de altíssimo nível por meio do conceito da utilização de funções da própria matemática na construção de programas no formato do cálculo *lambda*. Entre suas características mais relevantes, se destacam a ausência de efeitos colaterais e a avaliação preguiçosa.

2.2 Estratégias de Solução

Inicialmente, foram aplicadas as mesmas estratégias de solução utilizando as três linguagens, ou seja, foram implementadas recursões triviais para possibilitar aos programas o cálculo de forma incremental e recursiva do n-ésimo termo da sequência de Fibonacci. Dado o termo que se deseja calcular, é sempre chamada de forma recursiva o retorno da adição das duas funções, passando como parâmetro o termo (N) decrementado em uma e duas unidades, respectivamente. Este processo é repetido até o momento em que se tem o caso base, ou seja, os dois valores que somados equivalem ao primeiro termo da sequência (0 e 1). O modelo da declaração da função e da chamada recursiva são exibidos à seguir:

```
function fibonacci(n)
```

```
return fibonacci(n - 1) + fibonacci(n - 2)
```

Posteriormente, foi aplicada uma etapa de refatoração nas soluções. A principal motivação para a aplicação deste processo se deve às limitações presentes em algumas linguagens de programação (principalmente aquelas cujo paradigma é puramente funcional ou lógico). Entre elas podemos citar a menor eficiência na execução de programas e o consumo elevado de memória, aspectos provocados pelas recursões.

Todas as três implementações foram adaptadas, introduzindo recursões em cauda, ou seja, agora a chamada recursiva é a última operação à ser

realizada pela função. Isto elimina a necessidade de guardar informações referentes às chamadas anteriores da função na recursão, culminando em menor utilização da memória durante o processo de empilhamento (aliviando a *stack*).

Com isto em mente, a mesma ideia foi aplicada utilizando os três paradigmas de programação. A partir do caso base (0 e 1) que vamos denominar agora como (a) e (b), é chamada a função de forma recursiva de acordo com o termo desejado $N-1$ vezes, ou seja, a complexidade referente ao número de chamadas recursivas é relacionada diretamente com a magnitude do termo desejado da sequência ($\mathcal{O}(n)$).

A chamada recursiva sempre é a última instrução executada pela função, decrementando N em uma unidade e incrementando (a) e (b) gradualmente. O modelo da declaração da função e da chamada recursiva são exibidos à seguir:

function fibonacci(n, a, b)

return fibonacci(n - 1, b, a + b)

3 Resultados e Discussão

Nesta seção, serão discutidos de forma breve alguns aspectos relacionados aos resultados obtidos no desenvolvimento da solução para o problema, além de uma análise superficial em relação as vantagens e desvantagens de cada paradigma de programação explorado.

3.1 Implementação

Conforme citado anteriomente, a implementação da solução foi realizada utilizando três linguagens: *Python* (Figura 1), *Prolog* (Figura 2) e *Haskell* (Figura 3). O desenvolvimento foi auxiliado pelo uso da ferramenta de versionamento *GitHub*, e o código fonte está disponível publicamente em: `https://github.com/juliorodrigues07/fibonacci_clp`.

À seguir serão exibidos os *scripts* das soluções finais (com recursão em cauda) implementadas em cada uma das linguagens:

```python
def fibonacci(n, a = 0, b = 1):

    if n < 1 or type(n) is not int:
        return 'Deve ser informado um número inteiro e maior do que zero!'
    elif n == 1:
        return b

    return fibonacci(n - 1, b, a + b)
```

Figura 1: Implementação com Paradigma Imperativo (*Python*)

```prolog
fibonacci(N, Result) :- fibonacci_tail(N, 0, 1, Result).

fibonacci_tail(0, N, _, N).
fibonacci_tail(N, A, B, Result) :-
   N > 0,
   New_B is A + B,
   N1 is N - 1,
   fibonacci_tail(N1, B, New_B, Result).
```

Figura 2: Implementação com Paradigma Lógico (*Prolog*)

```haskell
fibonacci :: Int -> Int
fibonacci_aux :: Int -> Int -> Int -> Int

fibonacci n = fibonacci_aux n 0 1

fibonacci_aux n a b
    | n == 1 = b
    | otherwise = fibonacci_aux (n - 1) b (a + b)
```

Figura 3: Implementação com Paradigma Funcional (*Haskell*)

3.2 Análise

Embora todas as implementações tenham gerado código enxuto (problema de simples solução), os paradigmas distintos, somados a sintaxe distante entre as linguagens contribuíram para a construção de código bastante

distinto, mas que possuem a mesma ideia principal de solução em seu núcleo. Na próxima subseção, teremos uma visão mais detalhada sobre as particularidades de cada paradigma e linguagem correspondente.

Mas antes, vamos analisar de forma bastante superficial o desempenho de cada implementação em cada paradigma de linguagem, utilizando como métrica básica o tempo de execução. Para evitar viés em relação à execução em máquinas locais, todas as implementações foram testadas em *IDEs* online: Google Colab (*Python*), SWISH (*Prolog*) e Coding Rooms (*Haskell*).

Na implementação inicial, ou seja, utilizando a recursão comum, todos os *IDEs* encontraram problemas na execução, principalmente no caso da linguagem *Prolog*, cujos avisos de "estouro"de pilha (0.2 GB) começaram a se apresentar com termos da sequência de *Fibonacci* posteriores ao vigésimo, além do aumento exponencial no tempo de execução observado. Por isto, foi mantido o limite de N em 20 para executar os programas, obtendo resultados que são sumarizados na Figura 4:

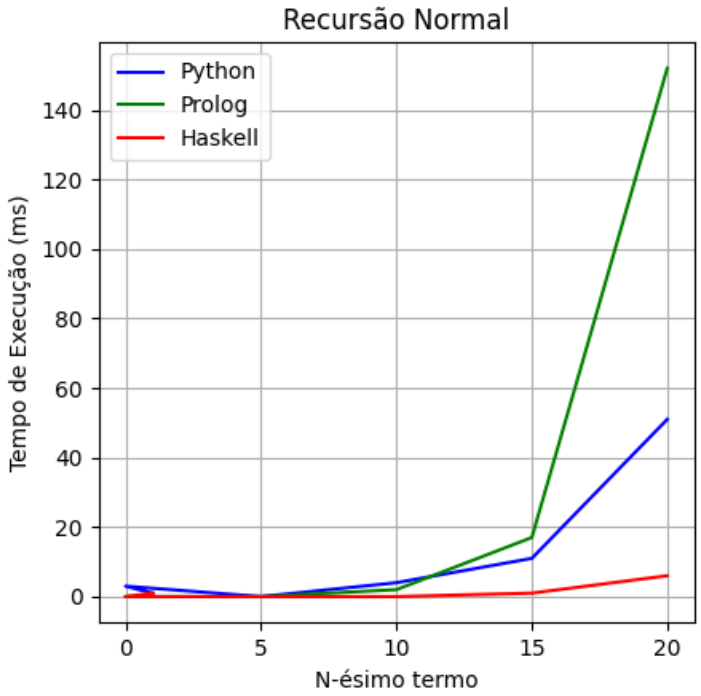

Figura 4: Tempos de execução na recursão normal

Na versão aprimorada da solução, ou seja, com a utilização da recursão em cauda, os resultados obtidos foram consideravelmente melhores, embora

uma limitação presente na *IDE* para *Haskell* tenha prejudicado os testes com termos de grande magnitude. Os maiores valores representáveis (*maxBound*) são da ordem de 2^{63}, o que corresponde à no máximo o 92º termo da sequência de *Fibonacci*, o que limitou os testes com N até este valor. Por isto, os resultados obtidos não são muito representativos (Figura 5), embora mostrem claramente que a recursão em cauda tem grande impacto no tempo de execução do algoritmo.

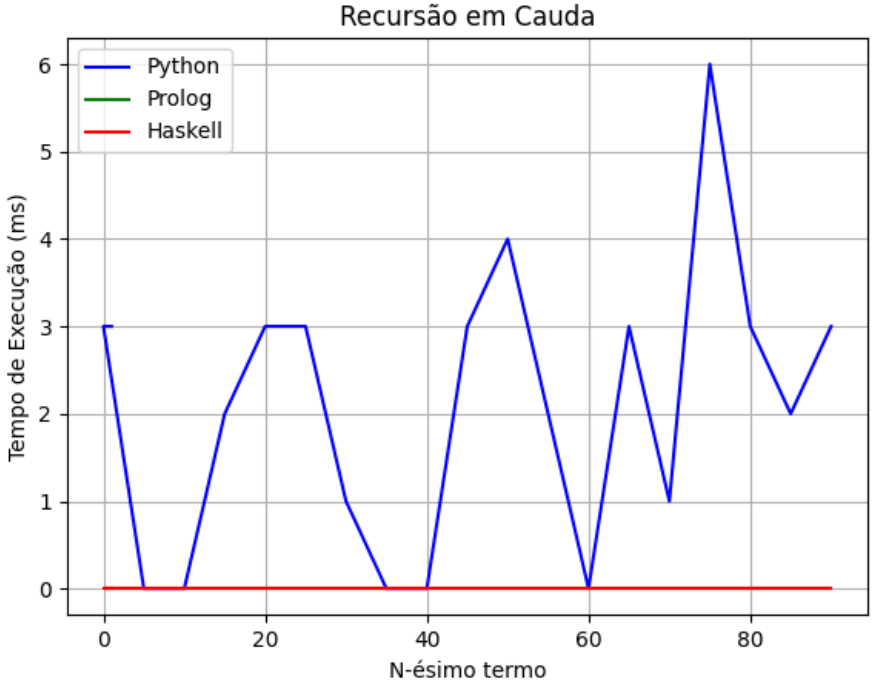

Figura 5: Tempos de execução na recursão em cauda

3.3 Vantagens e Desvantagens

A inclusão da recursão em cauda, embora possa trazer diversos benefícios para linguagens de programação em geral, julgo que prejudicou aspectos como legibilidade de código. Foi necessário incluir mais parâmetros, a fim de obter os resultados com base no incremento gradual dos valores base. No processo de refatoramento destas soluções, o paradigma em que encontrei maiores obstáculos na compreensão de como formular a solução foi o paradigma lógico, tanto devido à forma bastante distinta de pensar (fatos e regras), quanto à sintaxe da própria linguagem *Prolog*.

A solução implementada em *Haskell*, embora tenha gerado o código menos verboso, com abstrações de altíssimo nível, apresentou alguns empecilhos no desenvolvimento quando comparada à mesma escrita em *Python*. A linguagem *Haskell* não fornece suporte à inserção de parâmetros opcionais para suas funções, o que implicou na criação de uma segunda função auxiliar, que então realiza a chamada da função responsável por calcular o n-ésimo termo da sequência de *Fibonacci*, passando como parâmetros os valores base de a e b (0 e 1).

4 Conclusão

Neste trabalho, foram exploradas soluções para o cálculo do n-ésimo termo da sequência de *Fibonacci* que possuem a mesma ideia em seu núcleo, mas que foram implementadas de forma distintas, adequando-se aos paradigmas de programação presentes nas linguagens escolhidas, além é claro, de suas sintaxes.

Com este pequeno exemplo didático, podemos observar a exigência de diferentes formas de se pensar quando se deseja trabalhar com paradigmas de linguagens de programação distintos. Embora a maioria destas linguagens possam ser utilizadas para propósito geral, a escolha de uma linguagem para desenvolvimento deve levar em conta diversos fatores referentes aos paradigmas. Níveis de abstração, desempenho, legibilidade, e claro, a forma de raciocínio para a escrita de programas, são aspectos extremamente importantes que devem ser analisados extensamente no desenvolvimento de soluções para problemas de escopo geral.

Referências

- Optional Parameters in Haskell: `https://neilmitchell.blogspot.com/2008/04/optional-parameters-in-haskell.html`
- What is Tail Recursion: `https://www.geeksforgeeks.org/tail-recursion/`
- Introduction of Programming Paradigms: `https://www.geeksforgeeks.org/introduction-of-programming-paradigms/`
- Fibonacci Series - Recursive or Iterative? Elegance or Speed?: `https://groups.google.com/g/comp.lang.prolog/c/r_HO45eyVdc`
- Material disponível no portal didático na disciplina de CLP