CAPÍTULO 1

TODO Introdução

Programação é um trabalho majoritariamente artesanal. Essa é minha opinião e, sendo algo digno de debate, deixa uma margem para uma interessante discussão – por vezes divergente – com relação a este tipo de pensamento. Há aqueles que pensem no ato de escrever um programa de computador como algo majoritariamente mecânico: existiria um quê de criatividade, já que o programador precisa tomar as melhores decisões do ponto de vista do projeto de *software*; Mas, para tal, enumera-se ferramentas e processos que o auxiliam com planejamento e que, muitas vezes, acabam apontando, sem auxílio de "intuição", o melhor caminho a ser seguido nesse processo criativo.

Apesar de ver a pertinência do uso desses processos (como nos apresenta a disciplina de *Engenharia de Software*, por exemplo), ainda acredito na programação de computadores como uma *arte*, algo que envolve muito mais que apenas seguir práticas consolidadas.

Verna (2018, pp. 4) endossa o aspecto artístico na programação – que, por alguns, é considerado mera luxúria –, inclusive citando antigos apoiadores dessas ideias (como Knuth, Dijkstra e Ershov). Esses pensadores ilustres também buscavam clarificar que a descrição de um programa de computador envolve noções intrínsecas de *estética*, *beleza*, *estilo*, *prazer* e *emoção*, que tendem a pender muito mais para o lado artístico que o científico.

Pessoalmente, procuro pensar na escrita de um código de programação como sendo um processo muito similar à boa escrita em prosa de qualquer outro tipo de texto. O objetivo do autor é fazer-se entender através do que escreve; para tanto, é imperativo que palavras, orações e demais expressões gramaticais sejam ordenadas de forma inteligível, e o melhor recurso para garantir que o leitor compreenda o texto escrito é a leitura do mesmo, por parte do autor.

Ao escrevermos um programa de computador, realizamos um processo muito similar à composição ou redação. A escrita do código acaba, no fim das contas, por envolver três leitores: o *próprio autor*, um eventual *futuro leitor* do código (como um colega de trabalho) e a *máquina* em si (leia-se, o compilador ou interpretador, que fica efetivamente

responsável por "compreender" o código em questão e transformá-lo em linguagem de máquina).

O "leitor" mais fácil de agradar neste trio é a *máquina*, que não reclamará de aspectos como a estética do código, salvo quando programas como *linters* reforçam um certo estilo de escrita no processo de programação. Porém, a *máquina* ainda está suscetível a erros gramaticais ou de interpretação, provenientes de deficiências na escrita do código: estes poderão ocasionar tanto erros que antecedem a execução, por tratarem-se de código sintaticamente inválido, quanto a execução de algo não pretendido pelo programador – um erro semântico.

Aqui entra em ação outro potencial leitor do código: o próprio *autor*, o programador daquele segmento de código em si. O programador precisa verificar o código por erros, e também precisa criar correções. Se a forma como o código foi escrito não facilita o próprio trabalho daquele que o escreveu, então é sinal de que é necessário revisar a forma como o mesmo foi escrito. Escrever código sem preocupar-se com estética ou beleza é um erro comum de muitos programadores, que costumam ignorá-lo por não representar um problema em curto prazo. Mas, se aquele código precisar ser revisitado após algum tempo, o programador será o primeiro a sofrer com as consequências da ilegibilidade e/ou desorganização do próprio trabalho.

Por fim, temos o último tipo de leitor: o *futuro leitor* do código, que normalmente seria um terceiro; mas poderíamos pensar até mesmo no programador original, passado um bom tempo desde a última vez que viu o código: aos seus olhos, o programa ter-se-á tornado algo completamente desconhecido… um alienígena.

Aqui precisamos pensar no código como uma ferramenta *didática*. O código precisará ser dotado de uma simplicidade autoexplicativa. O *futuro leitor* precisará consertar algo no mesmo ou adicionar uma nova funcionalidade; para tanto, a forma de escrever o código será o melhor guia para deixá-lo a par dos passos a serem tomados, ou dos processos a serem seguidos para adicionar, remover ou modificar certas operações.

No espírito deste *aspecto didático* do código, que requer certa elegância e senso de beleza, é que escrevo este pequeno *software* em formato de livro. Escrever o interpretador de uma linguagem de programação e definir a especificação da mesma não é uma tarefa trivial, mas é uma tarefa *tangível*. Para provar este argumento e também para encorajar outros programadores a fazer o mesmo, escrevo aqui, detalhando passo-a-passo, todo o raciocínio por trás do planejamento para que se conceba uma linguagem de programação – como um caso especial, uma linguagem que seja um dialeto de Lisp.

A intenção não é apresentar um produto que seja extremamente polido e que não possa ser modificado posteriormente; antes, o *software*, assim como qualquer outro texto, é algo *vivo*, e pode inclusive ser modificado ou até mesmo "traduzido" para outras linguagens. Por isso, trabalho sob alguns pressupostos; estes serão enumerados mais tarde.

1.1 O que é Lisp?

Antes de mais nada, é essencial ressaltar que a linguagem que construiremos ao longo desse texto é um dialeto de Lisp.

CAPÍTULO 3

TODO Exemplos

3.1 Exemplo básico

```
;; -*- mode: lisp -*-

(defn square (x)
  (* x x))

(defn sum-of-squares (x y)
  (+ (square x)
     (square y)))

(defn gen-pairs elements
  (letfn ((gen-itr (lst)
            (if (or (nilp lst)
                    (nilp (cadr lst)))
                nil
                (cons (list (car lst)
                            (cadr lst))
                      (gen-itr (cddr lst))))))
    (gen-itr elements)))

(defmac squared-pair-sum elements
  `(map sum-of-squares
        (gen-pairs ,@elements)))

(squared-pair-sum 1 2 3 4 5 6 7 8 9)
```

Referências Bibliográficas

ABELSON, H.; SUSSMAN, G. J. **Structure and Interpretation of Computer Programs**. Cambridge: MIT Press, 1996. ISBN 978-0-262-51087-5. 4, 125

CHURCH, A. An unsolvable problem of elementary number theory. **American Journal of Mathematics**, v. 58, n. 2, p. 345–363, 1936. 3

IVERSON, K. E. Notation as a tool of thought. **Communications of the ACM**, v. 23, n. 8, p. 444–465, 1980. ISSN 0001-0782. 3

KNUTH, D. E. Literate programming. The Computer Journal, Stanford, CA, USA, v. 27, n. 2, p. 97–111, 1984. ISSN 0010-4620. 4

VERNA, D. **Lisp, Jazz, Aikido**: Three expressions of a single essence. 2018. Disponível em: <https://arxiv.org/abs/1804.00485>. 1