Sintaxe baseada em gramáticas de grafos para uma linguagem funcional visual voltada ao aprendizado de programação*

Marina Silva[1], **Ana Paula Lüdtke Ferreira**[1]

[1]Universidade Federal do Pampa
Campus Bagé

{marinasds2.aluno,anaferreira}@unipampa.edu.br

Abstract. *Imperative-based programming languages tend to emphasize syntactic aspects to the detriment of the problem-solving process. This work presents the syntax of the functional visual language Pandora, built from a graph grammar equipped with an algebra, whose rules emphasize the construction of a program with an emphasis on specification composition and reuse.*

Resumo. *Linguagens baseadas no paradigma imperativo tendem a enfatizar aspectos sintáticos em detrimento do processo de resolução de problemas. Este trabalho apresenta a sintaxe da linguagem visual funcional Pandora, construída a partir de uma gramática de grafos equipada com uma álgebra, cujas regras enfatizam a construção de um programa com ênfase em composição e reúso de especificações.*

1. Introdução

As primeiras experiências com programação no ensino superior costumam ser com linguagens de propósito geral fundadas no fluxo de execução do paradigma imperativo [Gomes e Mendes 2007, Pimenta 2019]. A complexidade das linguagens escolhidas e a forma como se representam os algoritmos em linguagens imperativas faz com que os métodos de ensino direcionem o foco (tanto do professor quanto do aluno) para a sintaxe da linguagem, em vez de destacar o processo de resolução de problemas por meio da programação, fazendo com que a atividade de desenvolvimento de algoritmos acabe com o foco na retirada de erros de compilação e não no pensar a solução de problemas.

A Engenharia de Software moderna preconiza o desenvolvimento modular e o reúso. O aproveitamento de especificações, modelos e códigos já construídos aumenta a produtividade das equipes de desenvolvimento e facilita os procedimentos de verificação, pelo uso de artefatos já testados, inclusive em ambiente de produção [Pressman 2016]. O reúso efetivo de artefatos de software exige do desenvolvedor um pensamento voltado à composição dos artefatos já conhecidos para a solução de um problema novo. A habilidade de compor especificações ou código não pode ser desenvolvida quando os estudantes, a cada nova tarefa, são instados a construir sempre uma solução a partir do zero.

Linguagens funcionais apresentam duas vantagens em relação ao ensino de algoritmos, nesse contexto. A primeira é que são focadas na solução de um problema, com definição das entradas e saídas, sem preocupação com detalhes de implementação e sem

*Trabalho de conclusão de curso de graduação, concluído.

desviar (muito) o foco do problema para a sintaxe. A segunda é que favorecem a ideia de que cada função deve ter um único propósito, devido à impossibilidade de efeitos colaterais gerados pelo programa, favorecendo modularidade e reúso de especificações. Como a noção de memória não está presente no paradigma, dificuldades com atribuições, alto acoplamento de módulos causados por variáveis globais e programação monolítica não ocorrem com facilidade.

Uma das formas de desenvolver habilidades de composição de artefatos de software é ensinar os alunos a programar por meio de linguagens que facilitem o uso de componentes já desenvolvidos para a construção de uma nova solução. O paradigma funcional de programação possui essa característica composicional [Sebesta 2018], ainda que as linguagens funcionais textuais possuam sintaxes que não favorecem o ensino. Cita-se, como exemplos, LISP [Touretzky 1990], Haskell [Thompson 1999], CLOS [Keene 1989], Elixir [Almeida 2018] ou Scala [Odersky et al. 2008].

Linguagens visuais já se provaram efetivas para aumentar o interesse de jovens pela área de programação [Resnick et al. 2009, Souza e Castro 2016, Feijó e De la Rosa 2016, Watanabe et al. 2020, Silva 2021]. A proposta a ser desenvolvida neste trabalho é reunir as vantagens da composicionalidade associada à programação funcional em uma linguagem visual que possa ser usada por iniciantes, com foco no processo de resolução de problemas, movendo a atenção do processo de compilação para o processo de desenvolvimento e execução de código. Espera-se, dessa forma, que os estudantes habituem-se a pensar em termos da solução de um problema como uma combinação da solução de problemas menores, favorecendo o pensamento lógico, o raciocínio abstrato, a modularidade e o reúso de especificações. Espera-se que o desenvolvimento dessas habilidades e postura frente à programação possa ser mantida quando os programadores migrarem para linguagens textuais com fluxo de execução imperativo, usadas em cursos de programação e na indústria de software.

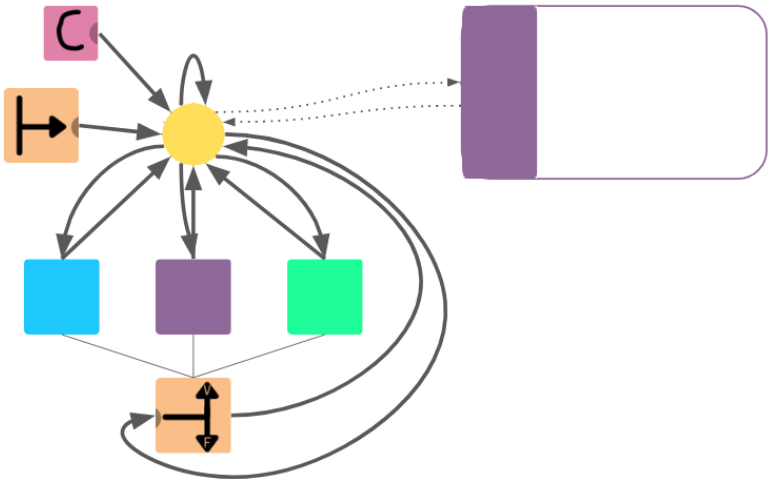
Este trabalho objetiva apresentar a sintaxe formal e a semântica informal da linguagem funcional visual Pandora, projetada para ser usada como primeira linguagem de programação em processos de ensino e aprendizagem em qualquer nível, incluindo o apoio ao ensino de Matemática no ensino fundamental. Essa é uma primeira aproximação do projeto da linguagem, que ainda não tem tipos de dados ou mesmo um interpretador construído. Contudo, tanto a sintaxe quanto a semântica de uma linguagem devem estar formalmente definidas para que se possa construir interpretadores corretos.

O restante do texto está organizado como se segue: a Seção 2 apresenta os princípios de construção e a sintaxe linguagem, a Seção 3 apresenta exemplos de programas em Pandora e a Seção 4 faz uma discussão sobre o trabalho e apresenta os desenvolvimentos futuros.

2. A linguagem Pandora

Um programa funcional é uma coleção de funções que podem ser definidas e invocadas em qualquer parte do código. Cada função recebe uma sequência de $n \geq 0$ parâmetros e devolve um único resultado. Não há operação de atribuição, o que elimina a possibilidade de efeitos colaterais e exige que o programador pense em termos de entradas e saídas. Programas em Pandora são expressos por um grafo. Definições e chamadas de funções, parâmetros e valores de retorno são representados por vértices. Constantes são considera-

Figura 1. Grafo tipo da linguagem Pandora



das funções de aridade zero e representadas com um símbolo especial. Arcos representam fluxo de dados ao longo do programa.

A sintaxe da linguagem Pandora é dada por uma gramática de grafos [Ehrig et al. 1996b] equipada com uma álgebra, para que a semântica de dados e operações possam ser definidas. Uma gramática de grafos é uma generalização das gramáticas de Chomsky [Lewis e Papadimitriou 1998] para reescrita de grafos. Formalmente, um grafo é uma tupla $G = (V, E, src, tar)$ em que V é um conjunto de nodos ou vértices, E é um conjunto de arcos ou arestas e $src, tar : E \to V$ são as funções que mapeiam as arestas em seus respectivos nodos de origem e destino. As regras da gramática são morfismos entre grafos. Um *morfismo de grafos* entre os grafos $G_1 = (V_1, E_1, src_1, tar_1)$ e $G_2 = (V_2, E_2, src_2, tar_2)$ é um par de funções $(f_V : V_1 \to V_2, f_E : E_1 \to E_2)$ tal que, para cada $e \in E_1$ tem-se que $f_V(src_1(e)) = src_2(f_E(e))$ e $f_V(tar_1(e)) = tar_2(f_E(e))$. Grafos podem ser rotulados ou tipados para que sua representação visual seja mais informativa. Um *grafo tipado* é uma tupla $G_T = (G, t, T)$ onde G e T são grafos e t é um morfismo total (i.e., ambas as funções do morfismo são totais) entre G e T.

O grafo tipo da linguagem restringe o tipo dos elementos e das relações que podem aparecer no programa e é apresentado na Figura 1. Os vértices do grafo em formato quadrado representam funções: verde e azul para as funções aritméticas e lógicas construídas na linguagem, roxo para funções definidas pelo usuário, rosa para funções constantes e salmão para a operação de teste condicional (*if-then-else*); o círculo amarelo representa um valor numérico ou lógico. Cada função é rotulada com um identificador, que não aparece no grafo por questões de simplicidade. O retângulo com um elemento roxo representa o escopo da definição de uma função, ao que os demais elementos estão ligados. Para visualização mais clara, os elementos do corpo da função aparecerão sempre em seu interior, estrutura garantida pelas regras. Note-se que o grafo tipo não restringe a quantidade de arestas entre os nodos. As restrições devem ser garantidas pelas regras da gramática.

O conjunto de regras que descreve a sintaxe da linguagem é apresentado a se-

guir. A abordagem algébrica *single-pushout* para gramática de grafos [Ehrig et al. 1996a] é usada neste trabalho, por razão de simplicidade da descrição das regras. Como não há deleção de elementos na construção de um programa representado por grafos, a abordagem *double-pushout* é igualmente adequada. A gramática da linguagem construída é composta por 20 regras para a construção dos programas, apresentadas na Figura 2. Além dos símbolos apresentados anteriormente, também é utilizado um quadrado cinza menor como símbolo não terminal, para garantir a definição de pelo menos uma função.

As regras (i) e (ii) permitem a definição de novas funções, no número que se desejar. Note-se que a estrutura para a definição de uma nova função contém somente o nome e um valor de saída. Os parâmetros e o corpo da função são construídos com outras regras. O símbolo f é uma variável da álgebra de strings usada para construção de identificadores. Os nodos referentes às álgebras não estão sendo mostrados por questão de simplicidade de notação. As regras (iii)-(v) inserem novos elementos na definição de uma função, respectivamente: um novo parâmetro, uma nova chamada de função e um novo parâmetro em uma chamada de função. Todos os elementos criados estão vinculados ao escopo da função f. As regras seguinte implementam a composição dos dados, seja estabelecendo ligações entre valores de parâmetros e chamadas de função (vi), indicando o valor de retorno da função sendo definida (vii), compondo duas funções (viii) ou compondo o resultado do ramo executado como entrada de outra função (ix). Note-se que todas essas regras possuem condições de aplicação negativas. Ou seja, uma valor somente pode ser passado a um nodo se ele já não estiver recebendo um valor de outra origem. A regra (x) estabelece que uma função pode ser substituída por uma função constante. No caso, o nodo referente ao conjunto suporte da álgebra também é omitido da regra. As regras (xi)-(xiv) determinam que a função usadas no grafo deve ser substituída por um operador aritmético e as regras (xv)-(xviii) fazem o mesmo para operadores lógicos. A regra (xix) cria uma estrutura do tipo *if-then-else* no corpo de uma função, em que o valor de saída da função (existente) a determina e execução de b ou se c consoante for verdadeiro ou falso, respectivamente. A junção dos valores de saída estabelece para onde vai o resultado, que pode ser proveniente de qualquer um dos ramos da função. A regra (xx) permite a composição de funções em um ramo de uma estrutura condicional.

3. Programas em Pandora

As regras da gramática permitem que um programa possa ser construído interativamente, com adição de novas funções, parâmetros em funções já existentes e novas chamadas no corpo de uma função, em um processo bastante similar ao usado em atividades de programação com linguagens textuais. A passagem de uma linguagem visual para uma linguagem textual exige um nível mínimo de similaridade no processo. Contudo, da mesma forma que em linguagens textuais, um problema somente pode ser executado se sua sintaxe está correta. Particularmente, a execução de um programa em Pandora exige que algumas condições sejam verdadeiras, especialmente as que se referem a passagens de parâmetros: (i) todas as chamadas de funções devem receber argumentos de fluxos de dados válidos e (ii) toda definição de função deve ter algum valor ligado ao seu valor de retorno.

O processo de derivação de um programa Pandora por meio da aplicação de regras reforça as noções de composição de funções, passagem de parâmetros e obtenção de resultados. Os programas resultantes são tais que o fluxo dos dados e das chamadas é

Figura 2. Gramática da linguagem Pandora

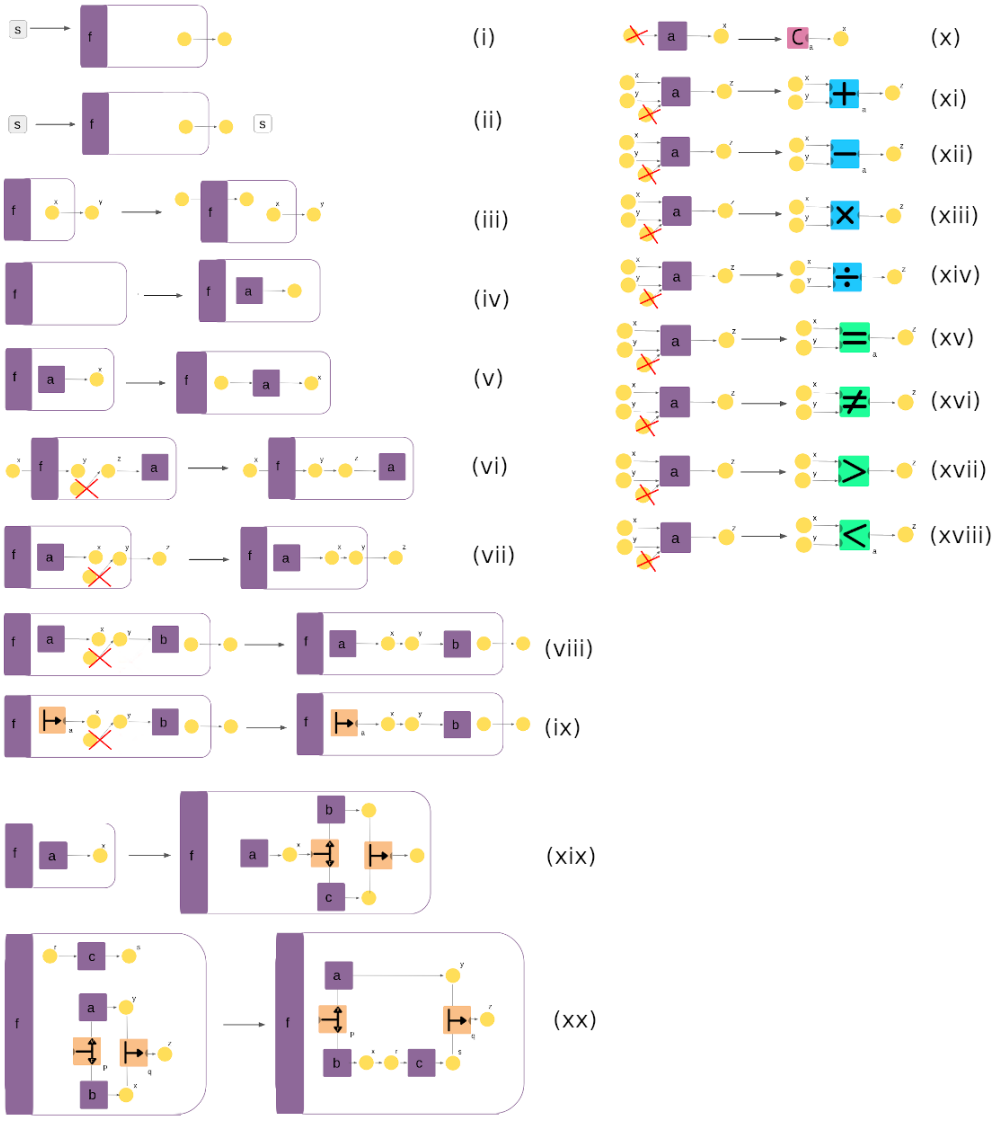


Figura 3. Função Pandora para adição e média de três valores

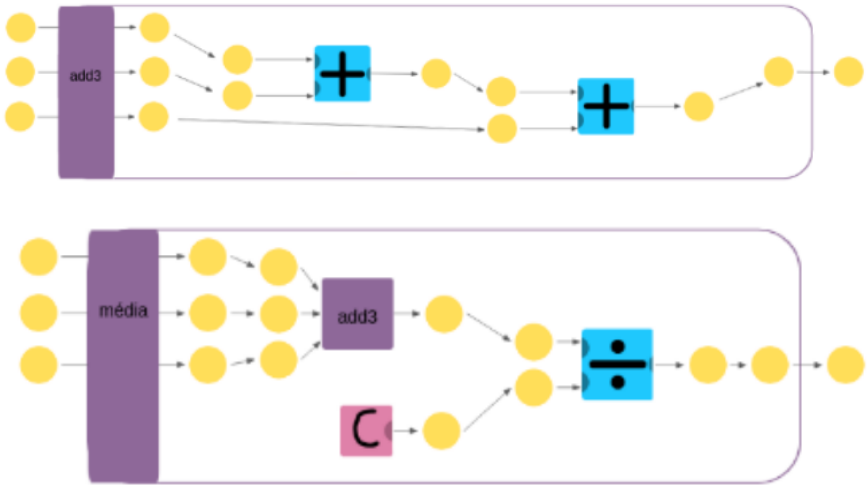
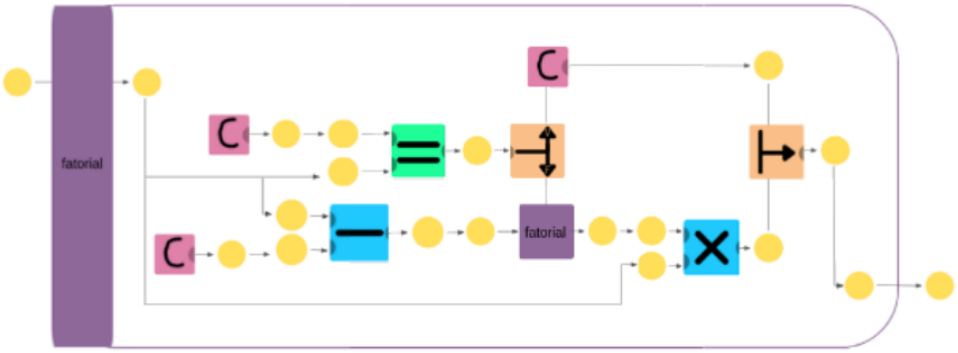


Figura 4. Função Pandora para o cálculo recursivo da função fatorial



tornado explícito. A Figura 3 apresenta a definição de uma função para o cálculo da adição de três valores (*add3*) e como essa função pode ser invocada por outra função que calcula a média de três valores (*média*), logo abaixo, na mesma figura.

A Figura 4 apresenta definição da função fatorial recursiva em Pandora, formalmente definida como

$$\text{fatorial}(n) = \begin{cases} 1 & n = 0 \\ n \cdot \text{fatorial}(n-1) & n > 0 \end{cases}$$

Note-se que não existem nomes de parâmetros em Pandora, ainda que eles possam ser incluídos sem esforço, fazendo uso da mesma álgebra usada para os nomes de funções. A função constante, neste ponto do desenvolvimento é somente indicada, mas espera-se implementar e mostrar o valor da constante, também por meio do uso de álgebras, que servirão para tipar os valores e encontrar erros em tempo de execução. A regra da composição permite que um mesmo valor seja argumento de diferentes funções, pontadas no grafo que representa o programa como setas com a mesma origem.

4. Conclusão

As construções da linguagem Pandora obedecem estritamente ao paradigma funcional, sem noção de memória ou de variáveis locais ou globais para armazenamento de resultados intermediários. Diferentemente das linguagens funcionais tradicionais, a notação da composição é dada por uma organização sequencial das funções, enfatizando a composição como um encadeamento de chamadas. O paradigma reforça a ideia de que cada função deve fazer uma única coisa e que funções podem ser combinadas para construir outras, acostumando o novo programador a pensar na questão do reúso de especificações. O foco da linguagem está em fazer o aprendiz pensar em termos de algoritmos como mapeamentos entre entradas e saídas, sem necessitar entender conceitos que têm a ver com implementações mas não com a resolução do problema proposto. Questões relacionadas à eficiência e ao armazenamento de resultados intermediários da computação podem ser codificados no interpretador de forma transparente ao usuário. A construção visual torna Pandora uma linguagem acessível para iniciantes em programação de diversas faixas etárias. A organização horizontal visa facilitar a transição para linguagens textuais.

Gramáticas de grafos são uma forma natural de representar estruturas visuais e sua sintaxe, formalmente. A semântica formal da linguagem Pandora está em construção. A partir da definição forma da semântica da linguagem, deverá ser construído um interpretador que possa ser executado sobre o grafo que representa o programa, enfatizando o fluxo de dados e o funcionamento das chamadas de funções. Já existe um proposta de IDE para a linguagem. A interface está sendo planejada de forma a facilitar as operações de encapsulamento de códigos em novas funções na edição de programas. A execução de um programa Pandora ma IDE, construída deverá mostrar os valores de parâmetros e resultados aparecendo nos círculos que correspondem ao fluxo de dados do programa.

Nesta definição inicial da sintaxe, somente funções numéricas e lógicas são suportadas. Pretende-se também, no futuro, expandir essa definição para construções que permitam tipos abstratos de dados e orientação a objetos. Para isso, gramáticas de grafos orientados a objetos e especificações algébricas deverão ser usados como suporte teórico.

Naturalmente que somente a definição sintática da linguagem não torna possível afirmar vantagens dessa abordagem sobre quaisquer outras, especialmente na questão do pensar computacional a longo prazo. Quando a IDE estiver completa, contudo, experimentos podem ser traçados de forma a mensurar – tanto em curto quando em médio prazo – o impacto do uso da linguagem no processo de solução de problemas por parte dos estudantes.

Referências

Almeida, U. (2018). *Learn Functional Programming with Elixir – New Foundations for a New World*. The Pragmatic Bookshelf.

Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., e Corradini, A. (1996a). Algebraic approaches to graph transformation. Part II: single-pushout approach and comparison with double pushout approach. In Ehrig, H., Kreowski, H.-J., Montanari, U., e Rozemberg, G., editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, chapter 4, pages 247–312. World Scientific, Singapore.

Ehrig, H., Kreowski, H.-J., Montanari, U., e Rozemberg, G. (1996b). *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1. World Scientific, Singapore.

Feijó, P. G. e De la Rosa, F. (2016). Roblock: Programming learning with mobile robotics. In *Proceedings of the ITiCSE '16*, page 361, New York, NY, USA. Association for Computing Machinery.

Ferreira, A. P. L. e Ribeiro, L. (2003). Towards object-oriented graphs and grammars. In Najm, E., Nestmann, U., e Stevens, P., editors, *International Conference on Formal Methods for Open Object-Based Distributed Systems, FMOODS*, volume 2884 of *Lecture Notes in Computer Science*, pages 16–31, Paris. Berlin: Springer-Verlag.

Gomes, A. e Mendes, A. J. (2007). Learning to program - difficulties and solutions. In *International Conference on Engineering Education (ICEE 2007)*.

Keene, S. E. (1989). *Object-Oriented Programming in Common Lisp*. Addison-Wesley.

Lewis, H. R. e Papadimitriou, C. H. (1998). *Elements of the Theory of Computation*. Prentice Hall, Upper Saddle River, 2nd edition.

Odersky, M., Spoon, L., e Venners, B. (2008). *Programming in Scala*. Artima, 4th edition.

Pimenta, J. M. M. (2019). Temple - uma linguagem de programação para o ensino de programação. Mestrado, Universidade de Évora, Évora.

Pressman, R. S. (2016). *Engenharia de Software: Uma abordagem profissional*. Bookman, Porto Alegre, 8 edition.

Resnick, M., Maloney, J. H., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K. A., Millner, A., Rosenbaum, E., Silver, J. S., Silverman, B. S., e Kafai, Y. B. (2009). Scratch: programming for all. In *Communications of the ACM*.

Sebesta, R. W. (2018). *Conceitos de linguagens de programação*. Bookman, Porto Alegre, 11 edition.

Silva, M. (2021). Proposta de uma linguagem composicional visual para ensino de programação. Trabalho de Conclusão de Curso (Graduação), Universidade Federal do Pampa.

Souza, S. S. e Castro, T. H. C. (2016). Investigação em programação com scratch para crianças: uma revisão sistemática da literatura. In *Anais dos Workshops do V Congresso Brasileiro de Informática na Educação (CBIE 2016)*.

Thompson, S. (1999). *Haskell: The Craft of Functional Programming*. Addison-Wesley, 2 edition.

Touretzky, D. S. (1990). *COMMON LISP: A Gentle Introduction to Symbolic Computation*. The Benjamin/Cummings Publishing Company, Inc., Redwood City.

Watanabe, T., Nakayama, Y., Harada, Y., e Kuno, Y. (2020). Analyzing viscuit programs crafted by kindergarten children. In *Proceedings of the ICER '20*, page 238–247, New York, NY, USA. Association for Computing Machinery.