Capítulo 2

Pensamento Computacional: Fundamentos e Integração na Educação Básica

Leila Ribeiro, Luciana Foss, Simone André da Costa Cavalheiro

Abstract

The goal of this paper is to support the realization of a course to clarify the meaning and discuss the relevance of Computational Thinking. The concepts are presented in an intuitive way, being illustrated with everyday situations. We differentiate logical from computational reasoning and discuss the importance of Computational Thinking in solving problems. The three pillars of Computational Thinking - Abstraction, Automation and Analysis - are presented, highlighting the role of each of them in developing the skills needed for the problem-solving process. The teaching of Computational Thinking is discussed based on the Computing Guidelines in Basic Education of the Brazilian Computer Society (SBC). Activities illustrating how these concepts can be developed in different school stages are presented.

Resumo

O objetivo deste texto é apoiar a realização de um minicurso para esclarecer o significado e discutir a importância do Pensamento Computacional. Os conceitos são apresentados de forma intuitiva, sendo ilustrados com situações do cotidiano. Diferencia-se o raciocínio lógico do computacional e discute-se a importância do Pensamento Computacional na resolução de problemas. Os três pilares do Pensamento Computacional - Abstração, Automação e Análise - são apresentados, destacando-se o papel de cada um deles no desenvolvimento das habilidades necessárias para o processo de solução de problemas. O ensino do Pensamento Computacional é discutido com base nas Diretrizes de Computação na Educação Básica da Sociedade Brasileira da Computação (SBC). São apresentadas atividades e realizadas dinâmicas para ilustrar como esses conceitos podem ser desenvolvidos em diferentes etapas escolares.

2.1. Introdução

Para entender o que é o *pensamento computacional*, precisamos entender o que é *computação*. Computação é uma ciência muito antiga na humanidade, embora só tenha sido reconhecida como uma área da ciência nas últimas décadas. Desde a antiguidade, babilônios e egípcios descreviam procedimentos de cálculos complexos (por exemplo, de navegação, geometria, astronomia) para que outras pessoas pudessem seguir essas instruções e assim usar o conhecimento e experiência dos cientistas mesmo sem serem especialistas. Essas pessoas eram os *computadores* de antigamente. Somente no final do século XX, o homem construiu as máquinas que chamamos hoje de computadores, com o objetivo de seguir instruções de forma mais rápida e sem erros. Mas, como se contrói o procedimento que será seguido por outras pessoas ou máquinas computadoras? O grande objetivo da Computação é *"compreender, formalizar e automatizar o raciocínio"*. Porém, diferente da Filosofia, aqui não estamos pensando de forma mais ampla sobre o raciocínio, mas sim interessados no processo de racionalização do raciocínio, ou seja, formalização do mesmo, o que permite a sua automação e análise (matemática).

A questão de formalização do raciocínio está intimamente relacionada à resolução de problemas. Para entender isso, tomemos como exemplo o raciocínio lógico. O objetivo do raciocínio lógico é basicamente encontrarmos (ou deduzirmos) verdades. O processo utilizado é, partido de premissas, que são fatos aceitos como verdades, utilizar regras bem definidas (do sistema lógico que se está usando) para encontrar novas verdades (conclusões). A dedução em si, que é a sequência de regras utilizadas, é comumente chamada de *prova* (de que a conclusão é verdadeira). O problema que está sendo resolvido é se uma sentença é ou não verdadeira: se encontrarmos uma prova a partir de sentenças que já sabemos que são verdadeiras confirmando a veracidade de uma nova sentença, ela será aceita como verdadeira. Podemos enxergar o raciocínio ou pensamento computacional como uma generalização do raciocínio lógico[1]: um processo de transformação de entradas em saída, onde as entradas e a saída não são necessariamente sentenças verdadeiras, mas qualquer coisa (elementos de um conjunto qualquer), sendo que as entradas e a saída nem precisam ser do mesmo tipo, e as regras que podemos utilizar não são necessariamente as regras da lógica, mas um conjunto qualquer de regras ou instruções bem definidas. Da mesma forma que o produto do raciocínio lógico é a prova, o produto do raciocínio computacional é a sequência de regras que define a transformação, que comumente chamamos de *algoritmo*. O problema que está sendo resolvido aqui é como transformar a entrada na saída. Exemplos concretos seriam: dado um número, como encontrar seus fatores primos? Dada uma pilha de provas de alunos, como ordenar essas provas? Dado um mapa rodoviário, como encontrar uma rota? Dados os ingredientes, como fazer um bolo? A Figura 2.1 mostra um esquema que sintetiza essa relação entre raciocínio lógico e raciocínio computacional.

Como o resultado do processo de raciocínio computacional deve ser uma descrição clara e não-ambígua de um processo, a Computação está fortemente baseada na Matemática, que provê uma linguagem precisa para descrição de modelos. Mas, diferente da Matemática, o objeto da Computação são os processos, ou seja, em Computação se constrói modelos de processos. Esses modelos, comumente chamados de *algoritmos*, podem

[1] A estreita relação entre provas e programas é conhecida como o isomorfismo de Curry-Howard (Wadler, 2015)

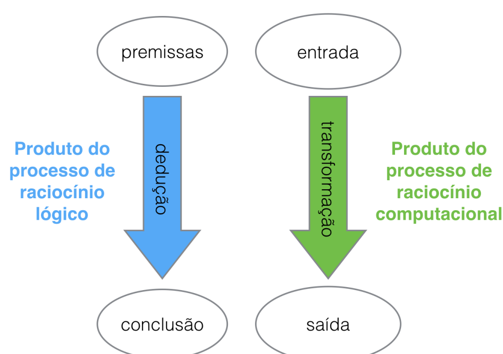


Figura 2.1: Raciocínio Lógico *versus* Raciocínio Computacional.

ser bastante abstratos, descritos em linguagem natural ou linguagens de especificação, ou programas em uma linguagem de programação. Um outro componente essencial da Computação é a representação da informação, pois grande parte dos processos que se quer descrever envolvem manipulação de estruturas complexas. Pode-se argumentar que na Matemática também usa-se diversas abstrações para nos ajudar a resolver problemas. Então por que precisamos de Computação? Somente para automatizar a solução do problema? Não, muito mais que isso. Vamos discutir esse ponto em um exemplo.

Imagine que o problema seja *"descreva o processo de ordenação de uma pilha de figurinhas"*. Essa não é uma tarefa trivial. A Matemática não nos ajuda a resolver esse tipo de problema, pois não provê as abstrações (de informação e processo) necessárias para descrever a solução. Além disso, não é objeto da Matemática investigar *Como construímos uma prova?*, ou mais genericamente, *Como construímos um algoritmo?*. A ênfase do raciocínio ou pensamento computacional não são apenas os produtos em si (provas ou algoritmos), e sim o *processo de construção desses produtos*, ou seja, além das abstrações necessárias para descrever algoritmos, o pensamento computacional engloba também técnicas para a construção de algoritmos, que podem ser vistas como técnicas de solução de problemas.

A evolução da Computação, em especial das áreas de Algoritmos, Teoria da Computação, Engenharia de Software e Ciência de Dados, descreve a trajetória da nossa aquisição de conhecimento com relação a como sistematizar (e se possível, automatizar) o *processo* de resolução de problemas (incluindo a construção de processos e representação da informação). Essa habilidade, de sistematizar, representar e analisar a resolução de problemas é chamada de *raciocínio ou pensamento computacional*[2] (Wing, 2006). A Figura 2.2 ilustra os pilares do *Pensamento Computacional*, que serão detalhados nas próximas seções.

A estrutura do texto de apoio ao minicurso está resumida na Figura 2.3.

[2] O termo *pensamento computacional* é uma tradução do termo original em inglês *computational thinking*. Embora do ponto de vista filosófico existam diferenças entre os termos *raciocínio* e *pensamento*, neste capítulo usaremos os dois termos como sinônimos.

Figura 2.2: Pilares do Pensamento Computacional. Fonte (Sociedade Brasileira de Computação (SBC), 2018).

1.1. Introdução
1.2. Solução de problemas e pensamento computacional
1.3. Abstração
  1.3.1. Abstrações para representar informação
  1.3.2. Abstrações para descrever algoritmos
  1.3.3. Técnicas para construir algoritmos
1.4. Automação
  1.4.1. Máquinas computadoras
  1.4.2. Linguagens para descrever algoritmos
  1.4.3. Modelagem computacional
1.5. Análise
  1.5.1. Viabilidade de encontrar solução computacional
  1.5.2. Correção e adequação das soluções
  1.5.3. Eficiência das soluções
1.6. Como ensinar Pensamento Computacional
  1.6.1. O Pensamento Computacional nas Diretrizes de Ensino de Computação na Educação Básica da SBC
  1.6.2. Exemplos de Atividades Práticas
Referências

Figura 2.3: Estrutura do texto de apoio ao minicurso.

2.2. Solução de problemas e pensamento computacional

O objetivo desta seção é definir o papel do pensamento computacional no processo de resolução de problemas. Para ilustrar, são apresentados problemas e discutidas diferentes soluções.

Problema 1

Ordene uma pilha com 10 figurinhas em ordem crescente. Considere que não há números repetidos e os números podem variar de 1 a 10.000.

Este problema pode ser solucionado facilmente por alunos que tenham aprendido a noção de ordem entre números. Como são apenas 10 figurinhas, normalmente eles apenas colocam todas na sua frente e vão buscando o próximo número para montar a pilha ordenada, fazendo as comparações necessárias mentalmente.

Problema 2

Ordene uma pilha com 1.000 figurinhas em ordem crescente.

Neste caso, apesar do problema ser essencialmente o mesmo (ou seja, é apenas uma nova instância do problema anterior), a solução *ad hoc* descrita acima é de difícil implementação porque quando colocamos 1.000 figurinhas na mesa fica difícil visualizar qual o próximo número. A estratégia mais usada neste caso é, antes de ordenar, dividir a pilha de acordo com a centena, e depois cada pilha de acordo com a dezena. Ou seja, dividir o problema em problemas menores até que a solução seja quase trivial. Depois das pilhas menores estarem ordenadas, precisa-se juntá-las de forma adequada para encontrar a solução do problema. Mas essa não é a única forma de resolver este problema, existem muitas outras, algumas mais e outras menos eficientes.

Problema 3

Descreva como ordenar uma pilha com figurinhas em ordem crescente.

Agora o problema é como descrever o método de ordenação, ou seja, o algoritmo utilizado para ordenar, para que o processo possa ser replicado, seguido por outras pessoas. Esta tarefa é bem mais difícil do que as anteriores, principalmente porque para descrever o processo de ordenação, precisamos falar sobre estruturas de dados (neste caso, uma pilha) e usar operações para definir o que deve ser feito em cada passo. Mas sem ter formalizados os conceitos de estruturas de dados e operações para definir processos (um algoritmo é uma descrição de um processo) é difícil solucionar este problema. Saber dar instruções de forma clara e precisa é uma habilidade necessária para todas as pessoas, e essa habilidade requer treinamento adequado. Além dos conhecimentos sobre os dados e como construir processos, precisa-se ter domínio da linguagem na qual os processos serão descritos. E a linguagem a ser usada depende de quem executará o processo: se for uma pessoa, pode-se usar linguagem natural, se for um computador, deve-se usar uma linguagem de programação. A grande diferença normalmente está no nível de abstração, pois linguagens naturais tendem a ser mais abstratas. Em breve discutiremos em mais detalhes a questão das linguagens.

Problema 4

O processo de ordenação descrito pode ser executado mais rápido se houverem mais pessoas para ajudar? Qual o número de pessoas seria o ideal? Existe alguma forma mais eficiente de ordenar as figurinhas? Dadas duas estratégias de ordenação, qual a melhor? O algoritmo descrito está correto, ou seja, no final da execução, a pilha está ordenada?

Solucionar este problema envolve analisar o algoritmo, ou método de ordenação, descrito na solução do Problema 3. Mas, mesmo que esse método esteja descrito de forma precisa, a análise da correção e eficiência do método não é uma tarefa trivial, apesar de ser de extrema importância porque um algoritmo que não gera o resultado desejado é inútil, bem como um que gera o resultado esperado, mas que demoraria demais para gerar este resultado (dependendo do problema e instância considerada, uma solução pode demorar vários anos para ser encontrada e esperar pode não ser viável do ponto de vista prático).

A seguir, vamos discutir três exemplos de soluções para o problema de ordenar uma lista de números em ordem crescente (uma abstração do problema de ordenar figurinhas) usando três (3) linguagens diferentes: uma visual, linguagem natural e linguagem de programação. O algoritmo apresentado é um algoritmo tradicional de ordenação chamado *quicksort* (Cormen et al., 2009). A Figura 2.5(a) mostra o algoritmo através de um diagrama. As setas representam o fluxo dos dados, as caixas vermelhas representam ações e a parte amarela pontos de decisão. A ideia é que a lista (a lista de números) entra na ação **ordena** , e então é testado se a lista está vazia ou não. Em caso positivo, o algoritmo termina retornando a própria lista vazia (que está ordenada pois não contém nenhum elemento). Em caso negativo, é identificado o primeiro elemento da lista (ação **primeiro**), e a lista é dividida em 2 partes, uma contendo os elementos menores que o primeiro (ação **seleciona-menores**) e outra contendo os elementos maiores que o primeiro (ação **seleciona-maiores**). Cada uma destas sublistas resultantes são ordenadas (repetindo o processo **ordena** para cada uma) e no final o resultado é construído juntando-se essas sublistas ordenadas e colocando o primeiro elemento no meio delas (ação **monta**). A linguagem visual é interessante porque deixa evidente o fluxo de dados e as ações envolvidas. Esse mesmo processo pode ser descrito em língua portuguesa (Figura 2.5 (b)). Note que a descrição do algoritmo em português é apenas uma frase, que foi quebrada em linhas diferentes na figura para maior clareza. É uma frase simples, mas descreve de forma sucinta e precisa o processo que deve ser seguido para ordenar a lista. Quando temos uma descrição precisa da solução, a implementação em uma linguagem de programação pode ser imediata: a Figura 2.5 (c) mostra como ficaria o programa correspondente descrito em uma linguagem funcional (Scheme ou Racket (*Racket Documentation*, n.d.)). Basicamente, o programa tem os mesmos elementos principais que a descrição textual, mas com uma sintaxe mais enxuta e rígida. Isso exemplifica um dos pontos que queremos enfatizar neste texto: *programar é fácil, o difícil é saber construir a solução dos problemas*. Se soubermos construir uma frase (ou texto) precisa em português, que descreve um processo, a programação seria um simples trabalho de tradução. Claro, dependendo da linguagem de programação utilizada a tradução pode ser mais fácil ou difícil, pois linguagens de programação diferentes oferecem

abstrações diferentes, mas ainda assim seria uma. A questão que realmente exige maior esforço e conhecimento é a construção da solução em si. E é esse o foco do *Pensamento Computacional*.

Os pilares do Pensamento Computacional (Wing, 2008), mostrados na Figura 2.4, provêm as habilidades necessárias para resolver os problemas citados anteriormente.

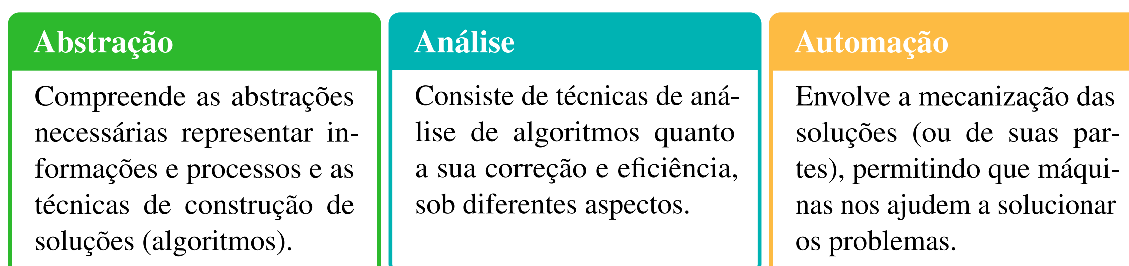


Figura 2.4: Pilares do Pensamento Computacional.

Em 2006, Wing (Wing, 2006) utiliza o termo *Pensamento Computacional* para apresentar a visão de que todas as pessoas podem se beneficiar do ato de pensar como um cientista da Computação. Informalmente, o pensamento computacional (Wing, 2011) descreve a atividade mental envolvida na formulação de problemas para admitir soluções computacionais e na proposta de soluções. As soluções (algoritmos) podem ser executadas por seres humanos ou máquinas, ou de maneira mais geral, por combinações de seres humanos e máquinas.

Já, em 1962, Alan Perlis (Perlis, 1962) argumentava que todos deveriam aprender a programar computadores no nível universitário. Ele identificou que a execução automatizada dos processos, explorada pela programação, mudaria a forma como os profissionais de todas as áreas pensariam sobre seu trabalho. No contexto da educação básica, na década de 1980, Papert (Papert, 1980) introduziu e popularizou a ideia de que computadores e o pensamento procedural poderiam afetar o modo como as crianças pensam e aprendem. Ao desenvolver o construcionismo (uma abordagem do construtivismo), defendia que o uso do computador (ou de ferramentas similares) na educação permitiria ao estudante desenvolver o seu raciocínio na solução de problemas e construir o seu próprio conhecimento.

O desenvolvimento do *Pensamento Computacional* não tem como objetivo direcionar as pessoas a pensarem como computadores. Ao contrário, sugere que utilize-mos a nossa inteligência, os fundamentos e os recursos da computação para abordar os problemas. Importante também observar que raciocinar computacionalmente é mais do que programar um computador. A Sociedade Internacional de Tecnologia em Educação (ISTE) e a Associação de Professores de Ciência da Computação (CSTA) (ISTE & CSTA, 2011) operacionalizaram o termo *Pensamento Computacional* como um processo de resolução de problemas.

2.3. Abstração

Nesta seção são apresentados os conceitos fundamentais necessários tanto para representar a informação quanto para descrever algoritmos. Também são discutidas as principais técnicas que podem ser usadas para construir algoritmos, ou seja, para auxiliar no

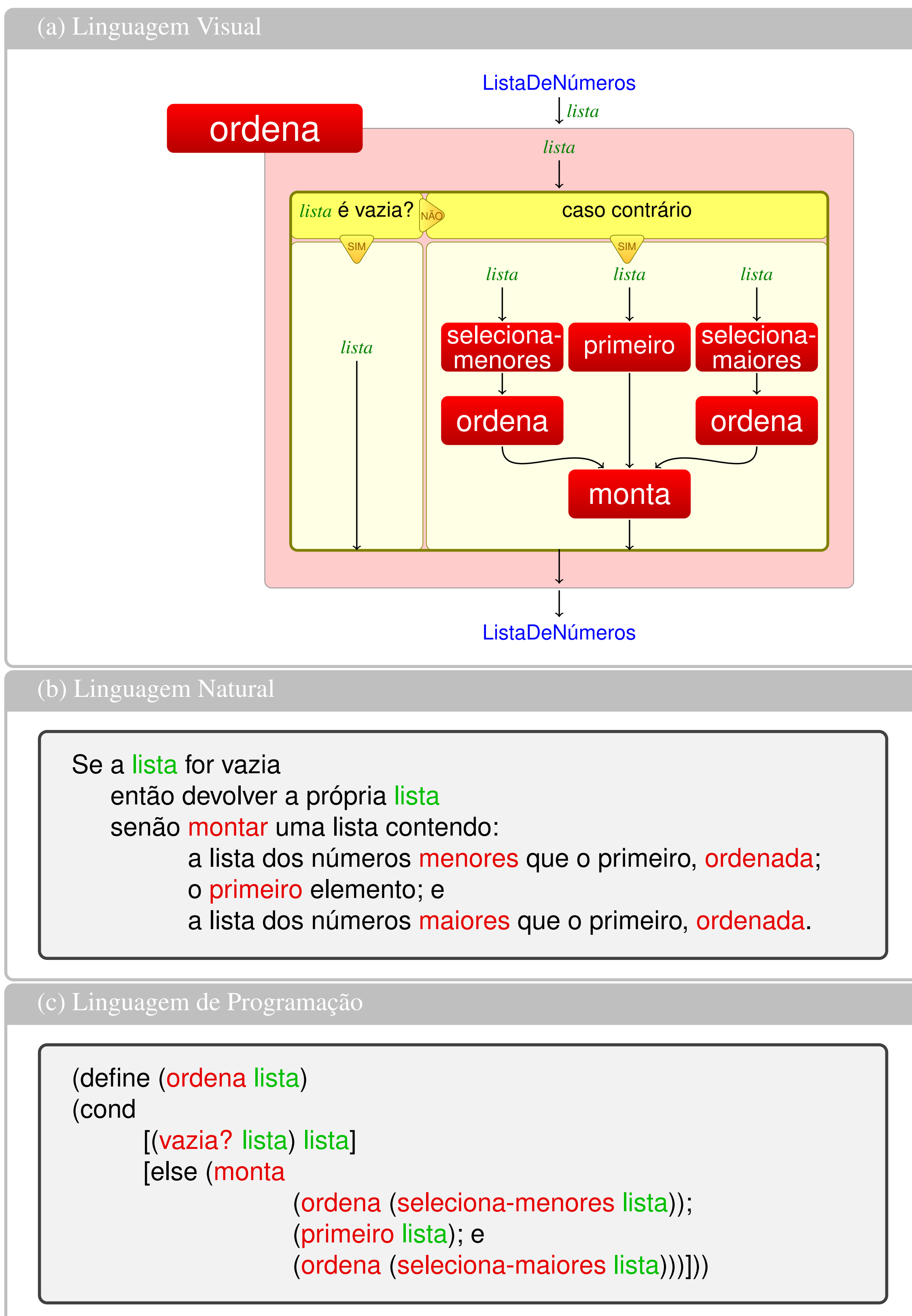


Figura 2.5: Três diferentes linguagens usadas na descrição do algoritmo Ordena

processo de resolução do problema.

A abstração (Wing, 2011; Barr & Stephenson, 2011; Wing, 2008, 2006) é um mecanismo importante no processo de solução de problemas, o qual consiste em simplificar a realidade, representando os aspectos mais relevantes de um problema e sua solução. Em Computação, a abstração permite que sejam construídos *modelos* de processos (naturais ou artificiais). Esses modelos envolvem tanto uma descrição da dinâmica do processo, ou seja, como ele evolui (através de algoritmos) quanto a descrição da informação que é tratada no processo (dado). Em Matemática, se usam abstrações para representar quantidades (que é um tipo de informação), que são os números. Para representar outros tipos de informação (muitas vezes bastante complexos) são necessárias abstrações adequadas. Para construir algoritmos, também é necessário que sejam conhecidas as abstrações que representam as operações fundamentais que nos permitem descrever processos. Finalmente, um ingrediente essencial deste eixo é aprender como se pode construir modelos (computacionais) da realidade. Juntamente com as abstrações para representar informações e processos, o domínio de técnicas de construção de algoritmos provê um subsídio fundamental para a habilidade de resolução de problemas.

2.3.1. Abstrações para representar informação

Se um algoritmo representa uma transformação de recursos (dados de entrada) em resultados (dados de saída), precisamos ser capazes de representar esses recursos e resultados de alguma forma. Como os algoritmos devem ser genéricos (ou seja, funcionar para várias entradas diferentes), a entrada e a saída devem ser representadas por conjuntos de elementos. Dependendo da finalidade do algoritmo, os elementos podem ser muito simples (um número, por exemplo), ou complexos (uma pilha de provas de alunos, um mapa, uma ficha de paciente de hospital etc). Para podermos descrever algoritmos necessitamos poder falar sobre esses dados, sejam eles simples ou complexos. E para isso precisamos das abstrações adequadas. Quando a entrada é um número ou uma palavra, podemos usar os conhecimentos já adquiridos durante anos em Matemática ou Português. Mas quando queremos processar uma pilha de provas para, por exemplo, ordenar de alguma forma, precisamos usar uma abstração para essa pilha. Quando queremos descrever como se encontra uma rota em um mapa rodoviário, precisamos falar do mapa e como ele é organizado para poder explicar para alguém como se procura um caminho. A diferença entre um número e uma pilha de provas é que o número representa um conceito indivisível, uma quantidade, enquanto a pilha é composta por unidades menores, que são as provas. E cada prova, por sua vez, pode ser composta de uma coleção de informações (nome do aluno, questões, respostas, nota). E para explicar como ordenar essa pilha de provas, precisamos acessar cada elemento da pilha, bem como as informações contidas em cada prova. Para descrever como encontrar uma rota em um mapa precisamos enxergar o mapa não como uma unidade, mas como um conjunto de cidades ligadas por estradas. Ou seja, para descrever algoritmos nós precisamos enxergar dados como composições de dados mais simples. Assim, temos vários níveis de abstração que podem ser usados para resolver um problema: no caso das provas, podemos enxergar a pilha como um todo, ou selecionar uma das provas, ou pegar uma informação de uma das provas. O nível de abstração escolhido dependerá do que se quer realizar em cada passo do algoritmo. Mas precisamos entender e poder falar sobre todos.

As abstrações de dados (chamadas estruturas de dados) mais importantes em Computação são (Cormen et al., 2009):

Registros : um registro representa uma coleção de informações de um objeto. Por exemplo, um registro de prova pode conter nome do aluno, questões, respostas, nota etc. Registros podem ser usados também para descrever dados de carteiras de identidade, formulários, cartão de respostas do vestibular etc;

Listas : uma lista é uma sequência de dados. Listas podem ser usadas como abstração para pilha de provas, baralho de cartas, cadeias de DNA, lista de compras, fila de banco, partituras (listas de notas musicais) etc;

Grafos : um grafo é uma estrutura que contém entidades (chamadas vértices) e relacionamentos (chamados arcos). Grafos podem ser usados para representar uma infinidade de estruturas, como redes sociais, mapas, árvores genealógicas etc.

Essas abstrações precisam ser trabalhadas de forma concreta e depois formalizadas, da mesma forma que o conceito de número na Matemática, para permitir que os alunos tenham capacidade de trabalhar sobre elas depois.

2.3.2. Abstrações para descrever algoritmos

Além de abstrações para dados, precisamos de técnicas para descrever as soluções em forma de algoritmos. Um algoritmo é composto por instruções que devem ser executadas de uma forma e na ordem definida para se atingir a solução desejada. Portanto, para se definir um algoritmo é necessário saber quais as instruções básicas que se pode usar, e quais operações podem ser usadas para se montar descrições dos procedimentos a partir dessas instruções básicas.

As instruções básicas dependem de quem vai ler o algoritmo. Se o leitor já sabe como ordenar uma lista, a instrução *"Ordene a lista"* é adequada. Caso contrário, precisa-se definir melhor como realizar esta instrução, através de instruções mais básicas que o leitor consiga entender. Em linguagens de programação, as instruções básicas são os comandos pré-definidos da linguagem, e existem bibliotecas de instruções que podem ser utilizadas. Para construir as soluções dos problemas para os quais não existem instruções básicas que os resolvam, usam-se operações que combinam instruções básicas de maneira a definir processos mais elaborados. As operações são basicamente de 3 tipos[3]:

Composição : permite juntar vários passos na descrição de um algoritmo. Esses passos podem ser conectados de várias formas diferentes (sequencial, paralela, por dependências etc);

Escolha : permite definir pontos de escolha em um algoritmo, que são momentos de decisão nos quais o próximo passo a ser executado depende da situação atual do processo;

[3]Essas operações foram vastamente estudadas pela área de álgebras de processos, que investiga as operações fundamentais para descrever processos(Milner, 1982; Hoare, 1985)

Repetição : permite que ações sejam repetidas em um algoritmo, de forma controlada. Existem várias formas de se definir como as repetições devem ser executadas (por exemplo, laços ou recursão).

Essas operações são implementadas de diversas formas em diferentes linguagens de especificação e programação.

2.3.3. Técnicas para construir algoritmos

Para se construir um algoritmo, não basta conhecer as abstrações de dados e processos. São necessárias técnicas que nos permitem chegar com mais facilidade do enunciado de um problema a uma solução. Entre estas técnicas, destacam-se:

Decomposição : É a técnica mais importante para se solucionar um problema, e consiste em decompor o problema em problemas menores, solucioná-los e combinar as soluções para obter a solução do problema original;

Generalização : É uma técnica que consiste em identificar padrões de comportamento e/ou dados e construir modelos genéricos que podem ser usados em vários contextos. Programas ou algoritmos são descrições de procedimentos, e podem ser usados como dados para outros programas ou algoritmos. Essa noção de que programas são dados, chamada de *meta-programação*, é um conceito fundamental da Computação e permite que se construam soluções extremamente elegantes, genéricas e simples para problemas complexos;

Transformação : Reutilizar e adaptar algoritmos é fundamental, e exige um grande poder de abstração. Muitas vezes problemas que, a primeira vista parecem totalmente diferentes podem ser solucionados pelo mesmo algoritmo fazendo-se apenas pequenas modificações. A técnica de transformação consiste em utilizar a solução de um problema para solucionar outro, através de transformação. Essas transformações podem ser feitas em diferentes contextos: para utilizar um algoritmo já existente para resolver o problema (reuso); para realizar melhorias em uma solução existente (refinamento); para adaptar soluções existentes a outras realidades (evolução); para compreender as relações entre problemas (redução) etc.

2.4. Automação

Nesta seção discute-se o que são "máquinas computadoras", ou seja, máquinas que seguem instruções, bem como o que são linguagens e a relação entre as linguagens para descrever algoritmos e as máquinas computadoras. Finalmente, fala-se sobre modelagem computacional.

A abstração nos permite encontrar e descrever um modelo de solução para um problema, e a automação é a mecanização de todas ou parte das tarefas da solução para resolver o problema usando computadores.

A automação (Wing, 2011; Council, 2010; Wing, 2008, 2006) envolve diferentes aspectos que devem ser levados em conta, que serão descritos nas subseções a seguir.

2.4.1. Máquinas computadoras

Para podermos automatizar a solução de um problema, primeiro é necessário saber se essa automatização é possível. Para que a mecanização seja possível, o computador deve ser capaz de interpretar as abstrações do modelo. Nesse contexto, um computador poderia ser um dispositivo mecânico, elétrico ou biológico (como por exemplo o DNA ou computadores moleculares) com capacidade de processamento, armazenamento e comunicação. Ou também poderia ser um humano, que segue fielmente os passos de um algoritmo, realizando o processamento de informações de forma mecânica. É importante saber escolher qual o tipo de computador (ou combinação de computadores) é o mais adequado para realizar uma tarefa desejada. Por exemplo, para preencher uma nota fiscal de venda, é melhor o vendedor preencher manualmente e fazer os cálculos no papel? Ou preencher manualmente e fazer os cálculos usando uma calculadora? Ou ainda, preencher usando um aplicativo de computador que fará os cálculos de forma automática? Para fazer a escolha adequada, é importante que se conheçam as características de cada máquina: para que elas servem, qual a dificuldade de utilizá-las, que tipos de problemas elas podem apresentar, como resolver esses problemas, etc.

2.4.2. Linguagens para descrever algoritmos

Escolhido o computador adequado, deve-se traduzir a solução do problema (algoritmo) para uma linguagem compreendida pelo computador. Cada tipo de computador reconhece uma (ou várias) linguagem(ns) diferente(s). Por exemplo, um computador tradicional compreende dados e instruções representadas por sequências de zeros e uns; o DNA compreende informações compotas por sequências de bases A (adenina), C (citosina), T (timina) e G (guanina); já um humano compreende sentenças descritas em diferentes linguagens naturais (português, inglês, espanhol, etc) e também linguagens formais como a matemática. Apesar de serem mais facilmente compreendidas, as linguagens naturais nem sempre são a melhor opção quando se quer descrever de forma precisa nossas abstrações. Uma linguagem natural é essencialmente ambígua e subjetiva, o que permite diferentes interpretações para uma mesma instrução. Já as linguagens compreendidas pelos computadores tradicionais usam uma representação bastante precisa. Existem diferentes linguagens que permitem descrições de soluções em diferentes níveis de abstração. A escolha da linguagem a ser utilizada, deve levar em conta essa característica. Quanto maior o nível de abstração, maior é a facilidade de descrever o algoritmo (solução do problema) nessa linguagem.

2.4.3. Modelagem computacional

A utilização de modelos pode auxiliar no entendimento de um problema, permitindo a simulação do comportamento dos sistemas envolvidos, bem como de soluções propostas. Os modelos podem ser físicos ou matemáticos. Por exemplo, pode-se construir modelos físicos de pontes que permitem medir deformações sofridas por tais estruturas ao receberem uma determinada carga; ou ainda, pode-se construir modelos matemáticos que podem ser simulados com o uso de um computador. A modelagem computacional fornece recursos para tratar problemas complexos e que envolvem um elevado número de variáveis. Para isso, é proposto o uso de métodos numéricos para tratamento do problema, associados a ferramentas computacionais e técnicas avançadas de programação. Exemplos de simulação

computacional podem ser encontrados nas mais diversas áreas como, no estudo de sistemas biológicos, no desenvolvimento de projetos e jogos, na previsão metereológica etc. O mercado está repleto de ambientes para simulação computacional que disponibilizam diversos recursos para construção de modelos de sistemas reais (Jahangirian et al., 2010) e a decisão de qual e como utilizar precisa ser tomada. Para isso, algumas considerações devem ser feitas: Como validar um modelo? Como tratar os resultados obtidos de uma simulação? Como saber se os resultados são válidos para o sistema real?

2.5. Análise

Esta seção apresenta e discute as principais formas de análise de problemas e suas soluções computacionais: como saber se é possível encontrar um algoritmo que resolve um problema, como saber se um algoritmo é realmente a solução de um problema, e como analisar a eficiência de algoritmos.

A Ciência da Computação provê fundamentos teóricos sólidos e uma rica teoria para análise e classificação de problemas (Cormen et al., 2009; Sipser, 2006; Lewis & Papadimitriou, 1997; Brassard & Bratley, 1996), permitindo descobrir se um problema tem ou não solução computacional, e também se pode existir algoritmo eficiente que o resolva, antes mesmo de tentar construir o algoritmo. A análise é de extrema importância pois fundamenta argumentação crítica sobre os problemas e suas soluções. De forma geral, a análise pode ser de três tipos, que serão discutidos a seguir.

2.5.1. Viabilidade de encontrar solução computacional

Nem todos os problemas podem ser resolvidos com o uso de computadores, existem vários problemas que não são passíveis de mecanização, chamados não-computáveis. Em alguns casos, apenas parte da solução pode ser executada por um computador. Por exemplo, não existe algoritmo que pode determinar se duas funções são equivalentes, mas é possível, dada uma entrada, verificar se duas funções produzem a mesma saída. Outros problemas não-computáveis são: determinar se um conjunto de dominós pode cobrir um tabuleiro; determinar se um algoritmo sempre termina; determinar se uma equação (polinomial) sempre tem uma solução (inteira); verificar se um programa tem vulnerabilidades de segurança etc.

2.5.2. Correção e adequação das soluções

Considerando os problemas que possuem soluções computacionais, como saber se um algoritmo proposto resolve o problema em questão? Uma solução está “correta” quando funciona exatamente como se espera em todas as situações. Se a solução foi dada por um programa de computador, afirma-se que o programa está “correto” quando ele fornece a saída esperada para todo valor possível de entrada. A questão é que o conjunto de todas as entradas possíveis para um programa, exceto para casos triviais, é extremamente grande. Ademais, os problemas que hoje em dia se apresentam nas mais diversas áreas do conhecimento possuem, em geral, soluções complexas.

Simulações e testes são algumas das técnicas utilizadas para encontrar erros e avaliar se os programas possuem características desejadas. Elas envolvem a execução de partes do programa ou de todo o sistema para avaliar propriedades de interesse, como por

exemplo, se o programa responde corretamente a determinadas entradas, se executa as principais funções dentro de um tempo aceitável, se atinge o resultado geral esperado, entre outros. Outra técnica de análise consiste na definição e utilização de modelos matemáticos para simular e verificar sistemas reais. A partir de uma especificação precisa (modelo matemático) é possível construir uma prova formal (utilizando de argumentação lógica e técnicas de demonstração) que garante que o modelo satisfaz determinadas propriedades. Diversas metodologias também já foram propostas para demonstrar que o projeto de um sistema está correto com respeito à sua especificação. Refinamentos (transformações precisas) podem ser especificados para transformar um modelo matemático em um projeto e um projeto em ma implementação, a qual, ao final do processo, é correta por construção.

2.5.3. Eficiência das soluções

A eficiência permite avaliar e comparar diferentes algoritmos quanto ao uso de recursos como tempo, memória, processador, energia, comunicação etc. Vamos supor que se precise avisar um colega que a reunião da tarde foi cancelada. Poderia-se enviar uma mensagem pelo celular para o colega. Alternativamente, poderia-se telefonar para ele. Também se poderia enviar um e-mail ou ir pessoalmente a sua casa. Há diversas alternativas para avisá-lo do cancelamento da reunião. Qual deve ser utilizada? Para escolher, pode-se levar em consideração os recursos disponíveis (por exemplo, celular, telefone), o tempo que será necessário para avisar usando cada alternativa, o custo de cada alternativa etc. Da mesma forma, existem diversos algoritmos que resolvem um mesmo problema. Para escolher qual utilizar em cada situação, precisamos poder analisar quantitativamente os algoritmos para dar subsídio ao processo de escolha da melhor alternativa.

Existem problemas para os quais não se encontrou até o momento soluções computacionais eficientes. Para esses problemas, chamados intratáveis, existem soluções (isto é, existem algoritmos que os resolvem), mas na prática levariam tanto tempo para chegar a um resultado (por vezes anos ou séculos, dependendo do tamanho da instância do problema) que se tornam inúteis. Já foi mostrado que muitos problemas de grande interesse prático se enquadram nessa categoria. É importante saber como identificar se um problema é intratável, para que não se tente encontrar solução eficiente para um problema que já foi classificado como intratável. Muitas vezes, pequenas modificações no enunciado do problema podem torná-lo tratável computacionalmente.

2.6. Como ensinar Pensamento Computacional

Esta seção aborda o ensino de Pensamento Computacional, relacionando os conceitos desenvolvidos nas seções anteriores com a proposta de Diretrizes de Ensino de Computação na Educação Básica da Sociedade Brasileira de Computação. São apresentadas atividades que ilustram como alguns conceitos fundamentais do pensamento computacional podem ser trabalhados .

2.6.1. O Pensamento Computacional nas Diretrizes de Ensino de Computação na Educação Básica da SBC

A proposta de Diretrizes de Ensino de Computação na Educação Básica foi elaborada por uma comissão de professores especialistas na área. Versões intermediárias foram

apresentadas e discutidas com a comunidade em eventos da SBC. Na elaboração, foram considerados os currículos de vários países, como Estados Unidos (ISTE & CSTA, 2011), Inglaterra (Guerra et al., 2012), Austrália (ACARA & NAP, n.d.), Alemanha (GOV.UK. Department for Education, n.d.), entre outros (CFE ARGENTINA, 2015; OPS, 2016), além de vários projetos-piloto existentes no Brasil (Bordini et al., 2016). O documento completo pode ser encontrado em (Sociedade Brasileira de Computação (SBC), 2018).

As Diretrizes da SBC englobam toda área de Computação, e estão organizadas em três eixos: Pensamento Computacional, Mundo Digital e Cultura Digital. A Computação auxilia no desenvolvimento de todas as dez competências gerais da Base Nacional Comum Curricular proposta pelo Ministério da Educação. O Pensamento Computacional auxilia no desenvolvimento de diversas competências gerais, mas com especial ênfase nas competências: 1 (compreensão do mundo); 2 (pensamento científico, criativo e crítico); 4 (comunicação); 5 (literacia digital); 7 (argumentação); e 9 (empatia e cooperação).

Para a apropriação dos conceitos fundamentais do Pensamento Computacional, a SBC sugere que eles sejam introduzidos do concreto ao abstrato. Ou seja, num primeiro momento os conceitos devem ser desenvolvidos a partir de situações do cotidiano e materiais manipuláveis para posteriormente introduzir suas respectivas formalizações e abstrações. Dessa forma, permite-se que o estudante, ao trabalhar com materiais concretos, crie modelos mentais que, em etapa posterior, servirão de base para que ele consiga abstrair e formalizar os conhecimentos. Por exemplo, nos Anos Iniciais pode-se trabalhar o conceito de lista e operações sobre listas utilizando-se baralhos, pilhas de figurinhas, fila de alunos, entre outros. Um segundo passo seria compreender a necessidade de descrever listas usando alguma linguagem, que até pode inicialmente ser uma linguagem visual, que é mais próxima da realidade concreta. Assim, quando este conceito for formalizado usando-se linguagens textuais e linguagens de programação, o passo de aprendizado que o aluno precisará dar será menor, pois ele já compreende o conceito em si, só precisa aprender a representá-lo de outra forma.

Nos Anos Iniciais os alunos já são expostos à noção básica de algoritmos quando, por exemplo, ensinam-se as operações aritméticas básicas. Essa etapa tem como objetivo que os alunos compreendam a necessidade de algoritmos para resolver problemas e compreendam a definição de algoritmos. Quer-se que eles identifiquem problemas cuja solução é um algoritmo, definindo-os através de suas entradas (recursos/insumos) e saídas (resultados) esperadas(os). Concomitantemente, são apresentados ao conjunto dos valores verdade e às operações básicas sobre eles (operações lógicas). Devem também entender que algoritmos são montados a partir de instruções e que a escolha da "máquina" define o conjunto de instruções que pode ser usado, impactando diretamente na montagem da solução. Deve ser enfatizada a diferença entre solução do problema e a representação da solução (usando alguma linguagem). Importante que identifiquem que a solução algorítmica de um problema envolve tanto a definição de dados (representações abstratas da informação) quanto do processo. A expectativa é que isso seja enfatizado, de forma que os estudantes tomem consciência da noção básica de algoritmo, sendo capazes de, a partir de conjuntos de instruções diversos, seguir e elaborar algoritmos para solucionar diferentes tipos de problemas, usando linguagem natural e linguagens pictográficas. Devem dominar as principais operações para a construção de algoritmos (composição sequencial, seleção e repetição) e ter noções de técnicas de decomposição de problemas. Além disso, espera-se

que os estudantes reconheçam a necessidade de representar e organizar a informação, cujos elementos podem ser atômicos (como números, palavras, valores-verdade) ou estruturados (como matrizes, registros, listas e grafos). Os estudantes devem dominar os conceitos dessas principais estruturas de dados, sendo capaz de identificar instâncias do mundo real e digital que possam ser por elas representadas.

Nos Anos Finais, espera-se que os estudantes sejam capazes de selecionar e utilizar modelos e representações adequadas para descrever informações e processos, bem como dominem as principais técnicas para construir soluções algorítmicas. Quanto à abstração de informação, devem reconhecer que entradas e saídas de algoritmos são elementos de tipos de dados. Apresenta-se o conceito de tipos de dados como conjuntos e as estruturas de dados, já introduzidas nos anos iniciais, são formalizadas. Além disso, devem conseguir descrever soluções, que possam ser automatizadas, de forma que máquinas possam executar partes ou todo o algoritmo proposto; construir modelos computacionais de sistemas complexos e analisar criticamente os problemas e suas soluções. Propõe-se inicialmente o uso de uma linguagem visual para descrever soluções de problemas, bem como o estabelecimento de relações de programas descritos na linguagem adotada com textos precisos em português. Posteriormente, pode-se fazer a transição para linguagens textuais de programação (lembrando sempre que a linguagem é apenas uma forma de representar soluções, o foco deve ser na construção da solução). Nessa etapa são também introduzidas e utilizadas as principais técnicas de solução de problemas: decomposição, generalização e transformação. E cada uma delas pode ser trabalhada em diferentes níveis e de forma gradual (Avila et al., 2019). A introdução à decomposição, num primeiro nível, envolve a utilização de uma (de)composição já estabelecida para resolver problemas. Os estudantes entram em contato com essa técnica utilizando uma estrutura previamente elaborada para resolver algum tipo de problema. Num nível intermediário, o problema a ser resolvido é apresentado decomposto e o estudante deve compor soluções de subproblemas para resolver o problema como um todo. Posteriormente, o processo de decomposição e composição deve ser construído pelo estudante. Já a generalização pode ser introduzida por meio da identificação de padrões e semelhanças em problemas, processos, soluções ou dados. E então, num nível intermediário, adaptam-se soluções ou partes de soluções para que elas se tornem mais genéricas. Por fim, criam-se modelos ou soluções que se apliquem a toda uma classe de problemas. Já dentre as técnicas de transformação, nos anos finais, o foco é dado a identificação de subproblemas comuns e ao reuso de soluções. Nos anos finais os alunos também são apresentados ao conceito de paralelismo, identificando partes de uma tarefa que podem ser realizadas concomitantemente e de recursão, identificando este conceito em diversas áreas (por exemplo, nas Artes, desenhando fractais a partir de regras ou em Literatura, construindo histórias dentro de histórias). Posteriormente, empregam o conceito de recursão para a compreensão mais profunda da técnica de solução por meio da decomposição de problemas.

No Ensino Médio a ênfase é na elaboração de projetos aplicando as diversas habilidades e conhecimentos adquiridos na etapa do Ensino Fundamental, e no desenvolvimento de habilidades relacionadas à análise crítica e argumentação, sob diferentes aspectos. No Ensino Medio são trabalhadas a técnica de transformação de problemas e o paradigma de metaprogramação (algoritmos que recebem outros algoritmos como entrada), que são conceitos necessários para a compreensão dos limites da computação, ou seja, dos limites

da formalização/racionalização. Esse entendimento, aliado aos fundamentos de inteligência artificial e robótica, provê a base necessária para uma discussão mais consubstanciada sobre o que é o Homem e o que é a Máquina, quais as similaridades e diferenças, não somente do ponto de vista físico, mas do ponto de vista filosófico, entendendo também as grandes questões éticas envolvidas na inteligência artificial. Outro conceito fundamental é a análise de algoritmos, tanto do ponto de vista de correção quanto de eficiência. Ao final do Ensino Médio, o aluno deve ter a habilidade de argumentar sobre algoritmos (processos), tendo meios de justificar porque a sua solução resolve de fato o problema, bem como analisar os tipos e quantidade de recursos necessários à sua execução.

2.6.2. Exemplos de atividades práticas

Nesta seção serão apresentadas atividades que permitem desenvolver habilidades relacionadas ao Pensamento Computacional. Algumas são atividades com materiais concretos, sem uso de ferramentas computacionais. Em outras usaremos o ambiente *WeScheme WeScheme Environment* (n.d.), no qual é possível executar programas na linguagem *Scheme*. Esta linguagem é uma linguagem construída especificamente para o ensino que segue a abordagem funcional e está inserida no contexto do projeto *Bootstrap.org*, que advoga que a ênfase do ensino deve ser em *projeto de algoritmos (ou programas)* e não em *programação*, pois é o projeto de algoritmos que desenvolve habilidades relacionadas à resolução de problemas, criatividade e novas formas de expressão. A ideia é que se use uma linguagem com sintaxe muito simples para que o aluno se concentre nos problemas a serem resolvidos e nas técnicas a serem usadas para resolvê-los, e não na linguagem em si. Idealmente, depois de compreender como construir as soluções (programas), o aluno poderia usar qualquer linguagem de programação de sua preferência. A linguagem *Scheme* é baseada na notação matemática, a seguir listamos as principais construções usadas (para uma introdução e vários exemplos do uso desta linguagem para ensino, ver *Bootstrap Program* (n.d.)):

Definição de constante: Para se dar um nome CONST a um valor val, ou seja, definir uma constante, a sintaxe é a seguinte:

(define CONST val)

Definição de função: Para se definir uma função com nome Fun, variáveis (parâmetros) n1 e n2 e corpo (expressão que define a relação entre o resultado da função e a entrada) expr, a sintaxe é a seguinte:

(define (Fun n1 n2) expr)

Aplicação de função: Para se aplica uma função com nome Fun aos argumentos (valores) val1 e val2, a sintaxe é a seguinte:

(Fun val1 val2)

A Figura 2.6 ilustra uma tela do ambiente *WeScheme* (que pode ser acessado no link *www.wescheme.org*). O lado esquerdo é chamado *janela de definições* e é onde constantes e funções são definidas, e o lado direito é a *janela de interações*, e é onde as funções são aplicadas. No exemplo da figura, são definidas as constantes CONSTANTE-DEZ (com

valor 10) e SOL, que é uma imagem de um círculo amarelo que tem 20 pixels de raio (a função circle é pré-definida na linguagem e gera a imagem do círculo). Também é definida a função soma-um, que soma uma unidade a um valor dado (em *Scheme* todas as funções são pré-fixadas, por isso a função + aparece antes dos valores aos quais ela está sendo aplicada, os valores x e 1). Na janela de interações, primeiro foi solicitada que o valor da constante SOL seja mostrando, e depois a função soma-um foi chamada duas vezes, com argumentos 2 e CONSTANTE-DEZ.

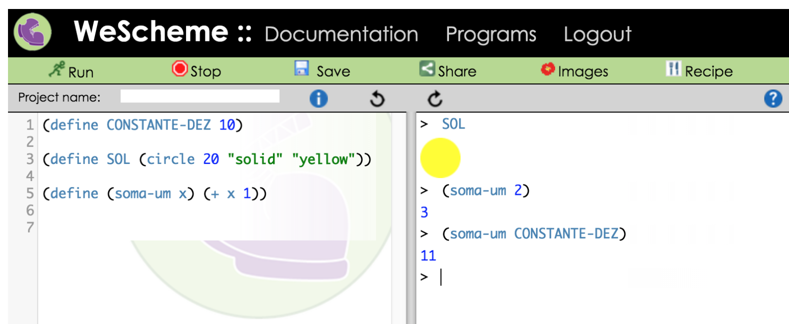


Figura 2.6: Tela do *WeScheme* (www.wescheme.org)

Cada atividade é apresentada em um quadro com sua descrição, objetivos, habilidades das diretrizes propostas pela SBC trabalhadas, materiais utilizados, metodologia e exemplo de avaliação. A Figura 2.7 apresenta uma breve descrição de cada atividade, e a descrição completa está no anexo A. Para a realização de cada atividade, espera-se que já tenham sido desenvolvidas as habilidades descritas em etapas anteriores nas diretrizes da SBC Sociedade Brasileira de Computação (SBC) (2018). Note que, apesar das habilidades estarem propostas para uma seriação específica nas diretrizes, não é imperativo que sejam desenvolvidas nessas etapas, apenas nesta ordem. Assim, em um contexto como atual, no qual a grande maioria dos alunos do Ensino Médio não desenvolveram as habilidades propostas para o Ensino Fundamental, é possível que elas sejam trabalhadas no Ensino Médio. As atividades a seguir são adequadas para alunos do Ensino Médio ou últimos anos dos Anos Finais.

2.7. Conclusão

A evolução da Computação, em especial as áreas de Teoria da Computação e Engenharia de Software, descrevem a trajetória da nossa aquisição de conhecimento com relação a como sistematizar (e se possível, automatizar) o processo de resolução de problemas. A metodologia utilizada para resolver problemas por meio de técnicas computacionais, bem como analisar criticamente as soluções, é chamada de Pensamento Computacional.

Neste capítulo foi apresentado o Pensamento Computacional e discutido sua importância na resolução de problemas. As principais etapas envolvidas no processo de solução de um problema delimitam os três pilares do Pensamento Computacional: Abstração, Automação e Análise. A primeira etapa consiste na busca de representações adequadas das informações e dos processos para descrever as soluções, bem como nas técnicas que nos permitem chegar com mais facilidade nessas soluções. A segunda etapa envolve a mecanização de (partes das) soluções, permitindo que máquinas nos auxiliem nas soluções do problemas. Já a terceira etapa compreende as técnicas necessárias para garantir que

Ordenação: Propõe-se a definição e simulação de um algoritmo de ordenação utilizando linguagem natural.

Entradas e Saídas: Identificam-se problemas cuja solução é um processo (algoritmo), definindo-os através de suas entradas (recursos/insumos) e saídas esperadas.

Construção de Imagens: Propõe-se a definição de algoritmos para construir bandeiras a partir de funções básicas para desenhar figuras.

Construção de Imagens no *WeScheme* : Propõe-se a tradução dos algoritmos de construção de bandeiras para a linguagem *Scheme* para execução no ambiente *WeScheme*.

Definição de Funções: Identificam-se partes de código a serem abstraídas, de forma a simplificar as soluções e constroem-se funções genéricas a partir da identificação de padrões nas definições de funções

Desenhando Cenas: Desenha-se cenas a partir de imagens, explicitando o processo de construção em si. No final, define-se uma forma de movimentação para um UFO e a cena é animada (com o UFO voando).

Figura 2.7: Lista de Ativades.

uma solução seja adequada para o propósito.

O Pensamento Computacional desenvolve a capacidade de compreender, definir, modelar, comparar, solucionar, automatizar e analisar problemas (e soluções) de forma metódica e sistemática, incluindo uma série de disposições e atitudes consideradas essenciais para os profissionais do século XXI (como pensamento crítico, colaboração, flexibilidade, adaptabilidade entre outras) ISTE & CSTA (2011). É possível resolver problemas de forma mais rápida e eficiente quando se usa o Pensamento Computacional. Assim como outras habilidades básicas (ler, escrever, falar), é necessário que esta metodologia seja trabalhada desde o Ensino Fundamental para permitir o seu pleno desenvolvimento. Neste capítulo, também é abordado o Ensino do Pensamento Computacional, relacionando os seus principais conceitos com a proposta de Diretrizes de Ensino de Computação na Educação Básica Sociedade Brasileira de Computação (SBC) (2018). Exemplos de atividades que ilustram como esses conceitos podem ser trabalhados são delineados no final do capítulo.

Diversas iniciativas vêm sendo tomadas para inserção do Pensamento Computacional no Brasil Ortiz & Pereira (2018), as quais variam bastante em sua abordagem, por exemplo, estimulando o seu desenvolvimento por meio da programação Zanetti et al. (2016); Rodrigues et al. (2015), robótica Costella et al. (2017); de Souza et al. (2016), jogos digitais Pessoa et al. (2017); Pinho et al. (2016) e/ou atividades desplugadas Silva Junior et al. (2017); *ExpPC - Explorando o Pensamento Computacional desde o Ensino Fundamental* (n.d.). Independente da abordagem, o importante é que os fundamentos da área sejam introduzidos numa profundidade compatível com cada etapa do ciclo escolar. As

Diretrizes para ensino de Computação na Educação Básica Sociedade Brasileira de Computação (SBC) (2018) apresentam competências específicas, objetos de conhecimentos e habilidades, norteando como essa inserção deve ser realizada.

Referências

ACARA, & NAP. (n.d.). *Australian curriculum.* (Disponível em `https://www.australiancurriculum.edu.au/`)

Avila, C. O., Foss, L., Bordini, A., & da Costa Cavalheiro, S. A. (2019). Evaluation rubric for computational thinking concepts. In *Ieee 19th international conference on advanced learning technologies icalt 2019.* (To appear.)

Barr, V., & Stephenson, C. (2011, February). Bringing computational thinking to k-12: What is involved and what is the role of the computer science education community? *ACM Inroads*, *2*(1), 48–54. Retrieved from `http://doi.acm.org/10.1145/1929887.1929905` doi: 10.1145/1929887.1929905

Bootstrap program. (n.d.). `https://www.bootstrapworld.org/`. (Acessed July 2019)

Bordini, A., Avila, C. M. O., Weisshahn, Y., da Cunha, M. M., da Costa Cavalheiro, S. A., Foss, L., ... Reiser, R. H. S. (2016). Computação na Educação Básica no Brasil: o Estado da Arte. *Revista de Informática Teórica e Aplicada*, *23*(2).

Brassard, G., & Bratley, P. (1996). *Fundamentals of algorithmics*. NJ, USA: Prentice-Hall, Inc.

CFE ARGENTINA. (2015). *Resolución no 263/15.* (Disponível em `http://www.unterseccionalroca.org.ar/imagenes/documentos/leg/Resolucion%20263-15%20%28Anexo%2001-%20Ley%20Nac%29.pdf`)

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms, third edition* (3rd ed.). The MIT Press.

Costella, L., et al. (2017). Construção de ambiente de ensino de robótica remota: democratizando o desenvolvimento do pensamento computacional em alunos da educação básica. In *Sbie* (pp. 354–363).

Council, N. R. (2010). *Report of a workshop on the scope and nature of computational thinking*. Washington, DC: The National Academies Press. Retrieved from `https://www.nap.edu/catalog/12840/report-of-a-workshop-on-the-scope-and-nature-of-computational-thinking` doi: 10.17226/12840

de Souza, I. M. L., et al. (2016). Explorando robótica com pensamento computacional no ensino médio: Um estudo sobre seus efeitos na educação. In *Sbie* (pp. 490–499).

ExpPC - Explorando o Pensamento Computacional desde o Ensino Fundamental. (n.d.). (Disponível em `wp.ufpel.edu.br/pensamentocomputacional`)

GOV.UK. Department for Education. (n.d.). *National curriculum in england: computing programmes of study.* (Disponível em `https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study`)

Guerra, V., Kuhnt, B., & Blöchliger, I. (2012). *Informatics at school - worldwide. an international exploratory study about informatics as a subject at different school levels* (Tech. Rep.). University of Zurich.

Hoare, C. A. R. (1985). *Communicating sequential processes*. NJ, USA: Prentice-Hall, Inc.

ISTE, & CSTA. (2011). *Computational thinking leadership toolkit.* (Disponível em `www.iste.org/docs/ct-documents/ct-leadershipt-toolkit.pdf`)

Jahangirian, M., Eldabi, T., Naseer, A., Stergioulas, L. K., & Young, T. (2010). Simulation in manufacturing and business: A review. *European Journal of Operational Research*, *203*(1), 1 - 13. doi: https://doi.org/10.1016/j.ejor.2009.06.004

Lewis, H. R., & Papadimitriou, C. H. (1997). *Elements of the theory of computation* (2nd ed.). Upper Saddle River, NJ, USA: Prentice Hall PTR.

Milner, R. (1982). *A calculus of communicating systems*. Berlin: Springer.

OPS. (2016). *Esi- ja perusopetuksen opetussuunnitelman perusteiden uudistaminen.* (Disponível em `https://www.oph.fi/ops2016`)

Ortiz, J. S. B., & Pereira, R. (2018). Um mapeamento sistemático sobre as iniciativas para promover o pensamento computacional. In *Sbie* (Vol. 29, p. 1093).

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. NY, USA: Basic Books, Inc.

Perlis, A. J. (1962). Computers and the world of the future. In (chap. The computer in the university). The MIT Press.

Pessoa, F. I. R., et al. (2017). T-mind: um aplicativo gamificado para estímulo ao desenvolvimento de habilidades do pensamento computacional. In *Sbie* (pp. 645–654).

Pinho, G., et al. (2016). Proposta de jogo digital para dispositivos móveis: Desenvolvendo habilidades do pensamento computacional. In *Sbie* (pp. 100–109).

Racket documentation. (n.d.). `https://docs.racket-lang.org/`. (Acessed July 2019)

Rodrigues, R., Andrade, W., & e Livia Sampaio, D. G. (2015). Análise dos efeitos do pensamento computacional nas habilidades de estudantes no ensino básico: um estudo sob a perspectiva da programação de computadores. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *26*(1).

Silva Junior, B. A., Cavalheiro, S. A. C., & Foss, L. (2017). A última árvore: exercitando o pensamento computacional por meio de um jogo educacional baseado em gramática de grafos. In *Sbie* (Vol. 28, pp. 735–744).

Sipser, M. (2006). *Introduction to the theory of computation* (Second ed.). Course Technology.

Sociedade Brasileira de Computação (SBC). (2018). *Diretrizes de ensino de computação na educação básica.* (Disponível em `http://www.sbc.org.br/educacao/diretoria-de-educacao-basica`)

Wadler, P. (2015, November). Propositions as types. *Commun. ACM*, *58*(12), 75–84. Retrieved from `http://doi.acm.org/10.1145/2699407` doi: 10.1145/2699407

Wescheme environment. (n.d.). `https://www.wescheme.org/`. (Acessed July 2019)

Wing, J. M. (2006, March). Computational thinking. *Communications of the ACM*, *49*(3), 33–35. doi: 10.1145/1118178.1118215

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, *366*(1881), 3717–3725. doi: 10.1098/rsta.2008.0118

Wing, J. M. (2011, March). Computational thinking–what and why? *The magazine of Carnegie Mellon University's School of Computer Science*.

Zanetti, H., Borges, M., & Ricarte, I. (2016). Pensamento computacional no ensino de programação: Uma revisão sistemática da literatura brasileira. *Brazilian Symposium on Computers in Education (Simpósio Brasileiro de Informática na Educação - SBIE)*, *27*(1).

Currículos resumidos

Leila Ribeiro

Possui Bacharelado e Mestrado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (1988 e 1991), e Doutorado em Informática pela Universidade Técnica de Berlim/Alemanha (1996). É professora titular da Universidade Federal do Rio Grande do Sul. Suas áreas de interesse são Fundamentos da Computação, Bioinformática e Educação em Computação. Em 1998, recebeu o Prêmio Santista de Informática (categoria Juventude). Coordenou e participou de diversos projetos de cooperação nacional e internacional. Ministra disciplinas de fundamentos de Computação há mais de 20 anos. Representa o Brasil como membro do TC1 (Fundamentos da Computação) da IFIP. É Diretora de Ensino de Computação na Educação Básica da SBC, sendo uma das autoras da proposta de Diretrizes para ensino de Computação na Educação Básica da SBC.

Luciana Foss

Possui Bacharelado em Ciência da Computação pela Universidade de Caxias do Sul (2000), Mestrado e Doutorado em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (2003 e 2008). É professora Associada da Universidade Federal de Pelotas. Tem atuado no ensino, pesquisa e extensão, tanto na graduação quanto na pós-graduação em Computação da UFPel. Suas áreas de interesse são Métodos Formais e Educação em Computação. Desde 2013, faz parte da equipe de um projeto intitulado "Explorando o Pensamento Computacional para Qualificação do Ensino Fundamental", o qual integra ensino, pesquisa e extensão (trabalhando em propostas de atividades e metodologias para desenvolver o Pensamento Computacional no Ensino Fundamental).

Simone André da Costa Cavalheiro

É Engenheira Civil pela Universidade Católica de Pelotas (1998), Licenciada em Matematica pela Universidade Federal de Pelotas (1998), possui mestrado (2000) e doutorado (2010) em Ciência da Computação pela Universidade Federal do Rio Grande do Sul. É professora associada da Universidade Federal de Pelotas (UFPel). Atua na área de Fundamentos da Computação e Educação em Computação, principalmente em Métodos Formais e Pensamento Computacional. É coordenadora do projeto "Explorando o Pensamento Computacional para Qualificação do Ensino Fundamental", o qual integra ensino, pesquisa e extensão, com o apoio da Secretaria Municipal de Educação e Desporto de Pelotas. É membro da Comissão de Educação Básica da SBC. tendo contribuído com a proposta de Diretrizes para ensino de Computação na Educação Básica.

A. Exemplos de Atividades

ATIVIDADE DE ORDENAÇÃO

DESCRIÇÃO GERAL

Propõe-se a definição e simulação de um algoritmo de ordenação utilizando linguagem natural.

OBJETIVO

Introduzir as noções básicas de algoritmos, enfatizando a importância da descrição precisa de instruções em uma solução algorítmica, de forma que seja compreensível pela pessoa ou máquina que irá executá-la.

HABILIDADES TRABALHADAS

- Compreender a definição de algoritmos resolvendo problemas passo-a-passo;
- Compreender a necessidade de algoritmos para resolver problemas;
- Definir e simular algoritmos (descritos em linguagem natural ou pictográfica) construídos como sequências e repetições simples de um conjunto de instruções básicas.

MATERIAIS UTILIZADOS

Cartões com números (o número de cartões depende da faixa etária, para ensino médio em torno de 200 números para cada grupo).

METODOLOGIA

Passo 1: São propostas as tarefas descritas a seguir, a serem realizadas em grupos:

Tarefa1: Ordenar uma pilha de números recebidas.

Tarefa 2: Descrever os passos necessários para fazer a ordenação da pilha de números (conforme a estratégia utilizada na tarefa anterior).

Tarefa 3: Ordenar a pilha seguindo as instruções recebidas dos colegas. Seguir estritamente o que está escrito, sem fazer perguntas para o grupo que elaborou as instruções.

Passo 2: Discussão e avaliação.

AVALIAÇÃO

Discutir as seguintes questões, avaliando a compreensão dos alunos:

1. O que é mais fácil: ordenar, seguir instruções ou elaborar as instruções? Por quê? *Obj.: Diferenciar estas três atividades e analisar o tipo de habilidade necessária para realizar cada uma - compreensão/elaboação de texto, criatividade, etc.*

2. Qual o critério de ordenação utilizado? Existem outros? Dê 2 exemplos. *Obj.: Perceber que existem diferentes ordenações possíveis.*

3. Foi possível ordenar a pilha seguindo as instruções dos colegas? Um aluno dos anos iniciais conseguiria fazer isso? O que ele precisa saber para conseguir seguir as instruções? *Obj.: Identificar as habilidades necessárias para escrever ou seguir instruções.*

4. Com relação às instruções propostas pelo seu grupo, quantas pessoas trabalharam na ordenação? Havia tarefas para todas? Se o grupo fosse menor, afetaria o tempo necessário para ordenar? E se fosse maior? *Obj.: Perceber que existem diferentes algoritmos para executar uma mesma tarefa, que podem envolver graus de paralelismo distintos, que o paralelismo pode levar a tempos de execução menores, mas que existe um limite para essa redução do tempo de execução.*

5. Você acha que as instruções propostas pelo seu grupo funcionam para ordenar qualquer pilha de números? Existe alguma restrição (por exemplo, só funciona se os números forem de 1 a 1500, só se o grupo for de no mínimo N pessoas, só se não tiverem "centenas" faltando ...)? *Obj.: Identificar limites dos algoritmos propostos.*

6. E se a gente quisesse ordenar uma lista de palavras, as instruções seriam muito diferentes? O que seria necessário mudar? *Obj.: Perceber que o mesmo tipo de processo (algoritmo) pode ser usado sobre dados diferentes. Identificar que, no caso da ordenação, a operação de comparação entre dois itens é o que muda, pois esta depende do dado que é ordenado.*

Atividade de Entradas e Saídas

Descrição Geral

Identificam-se problemas cuja solução é um processo (algoritmo), definindo-os através de suas entradas (recursos/insumos) e saídas esperadas.

Objetivo

Apresentar a definição de problema como uma relação entre insumos e resultado, destacando que entradas e saídas de algoritmos são elementos de tipos de dados.

Habilidades trabalhadas

- Compreender a definição de problema como uma relação entre entrada (insumos) e saída (resultado), identificando seus tipos (tipos de dados, por exemplo, número, string, etc.);
- Reconhecer que entradas e saídas de algoritmos são elementos de tipos de dados;
- Formalizar o conceito de tipos de dados como conjuntos.

Materiais utilizados

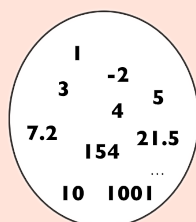
Folha com exercícios de avaliação.

Metodologia

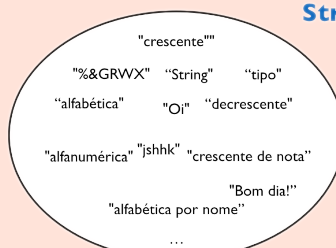
Passo 1: Retoma-se o problema da ordenação, trabalhado na atividade anterior, definindo-o em termos das entradas e saídas esperadas, identificando os seus tipos.

Passo 2: Introduz-se o conceito de tipos de dados, exemplificando com diversos tipos primitivos e propõem-se exercícios.

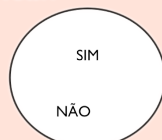
Number



String



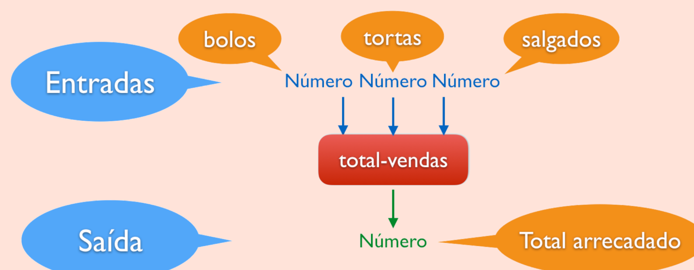
Boolean



Image

Tarefa: Definir o problema a seguir em termos de seus tipos de entradas e saídas.

"O dono de uma confeitaria vende bolos, tortas e salgados. Cada bolo é vendido por R$ 3,00, cada torta sai por R$ 5,00 e cada salgado custa R$ 1,50. Ao fim do dia, a contabilidade é realizada por 2 funcionários. Cada um destes funcionários recebe um salário de R$ 20,00 por hora. O dono da confeitaria deseja saber qual foi o total arrecadado no dia."



Passo 3: Avaliação.

Avaliação

Solicitar a resolução dos seguintes exercícios e na sequência corrigi-los.

1. Nos ítens a seguir, dê um nome para a solução (algoritmo) e indique os tipos das entradas e saídas.

   (a) O banco do Luís dá um lucro de 10% por cada ano que ele deixa seu dinheiro depositado na poupança. Para ajudá-lo a decidir quanto ele deve depositar, ele gostaria de um algoritmo que, dado um valor, diga quanto ele terá ao final de um ano.

   (b) Um jogador tem várias cartas na mão. Ele gostaria de saber quantas cartas de ouros ele tem.

   (c) Um jogador tem várias cartas na mão. Ele gostaria de saber se ele tem alguma carta de ouros.

   (d) Um grupo de amigos se juntou para comprar um presente para um outro amigo. Eles querem saber quanto cada um vai precisar gastar.

   (e) A temperatura no Brasil é medida em graus Celsius (C) e nos Estados Unidos em graus Fahrenheit (F). Existe uma relação entre essas medidas, dada pela fórmula:

   $$\frac{C}{5} = \frac{F-32}{9}$$

   Ana gostaria de ter um algoritmo para transformar graus C em F e F em C.

   (f) João gostaria de encontrar um caminho em um mapa.

   (g) Márcia quer um bolo.

Obj.: Nomear adequadamente o algoritmo, bem como identificar as quantidades e tipos de entradas e saída.

Atividade de Construção de Imagens

Descrição Geral

Propõe-se a definição de algoritmos para construir bandeiras a partir de funções básicas para desenhar figuras.

Objetivo

Apresentar os mecanismos de composição e decomposição, utilizando-os de forma empírica na solução de problemas.

Habilidades trabalhadas

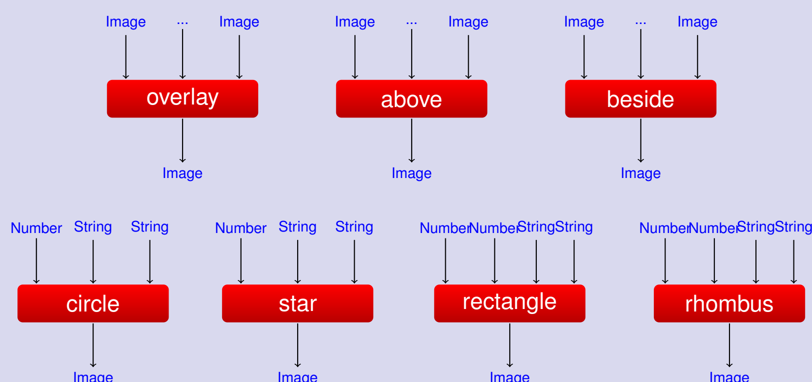
- Identificar problemas de diversas áreas do conhecimento e criar soluções usando a técnica de decomposição de problemas;
- Utilizar uma linguagem visual para descrever soluções de problemas envolvendo instruções básicas de processos.

Materiais utilizados

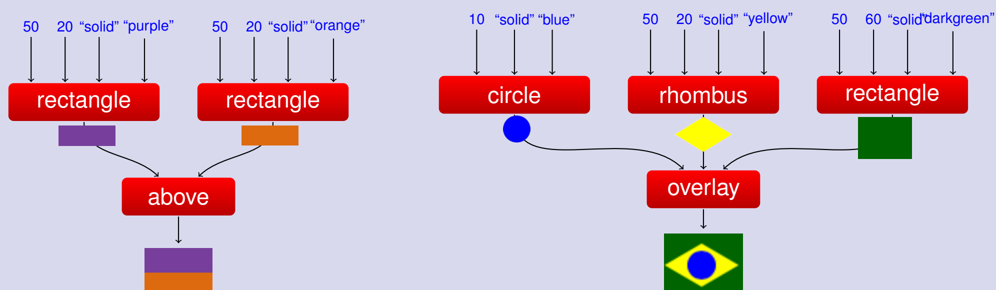
Folhas A3, cartões auto-colantes com os nomes das funções, canetas.

Metodologia

Passo 1: Apresentam-se as funções básicas a seguir para desenhar figuras, discutindo as entradas e os resultados esperados.



Passo 2: Exemplifica-se a montagem de bandeiras com as funções apresentadas.

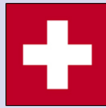


Passo 3: Avaliação

AVALIAÇÃO

Solicitar a resolução dos seguintes exercícios e na sequência corrigi-los.

1. Utilizando as funções básicas para desenhar figuras, monte algoritmos para construir as bandeiras abaixo:



2. Monte um algoritmo para construir uma bandeira para seu grupo. Em uma folha separada, mostre o resultado da execução deste algoritmo.

Obj.: Identificar as figuras básicas que compõem cada bandeira e juntá-las de forma que a solução leve ao resultado desejado.

ATIVIDADE DE CONSTRUÇÃO DE IMAGENS NO *WeScheme*

DESCRIÇÃO GERAL

Propõe-se a tradução dos algoritmos de construção de bandeiras para a linguagem *Scheme* para execução no ambiente *WeScheme*.

OBJETIVO

Formalizar o conceito de função e composição de funções. Relacionar algoritmos descritos em linguagem visual com programas na linguagem *Scheme*.

HABILIDADES TRABALHADAS

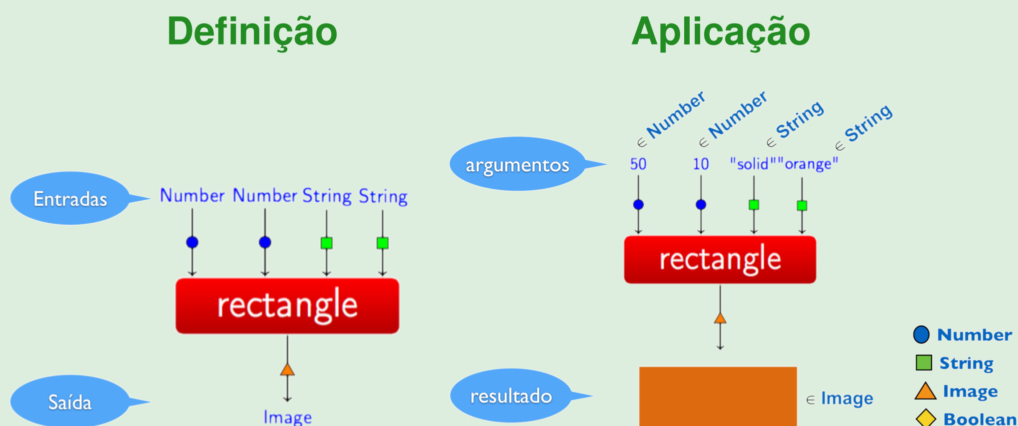
- Utilizar uma linguagem visual para descrever soluções de problemas envolvendo instruções básicas de processos;
- Relacionar programas descritos em linguagem visual com textos precisos em português.

MATERIAIS UTILIZADOS

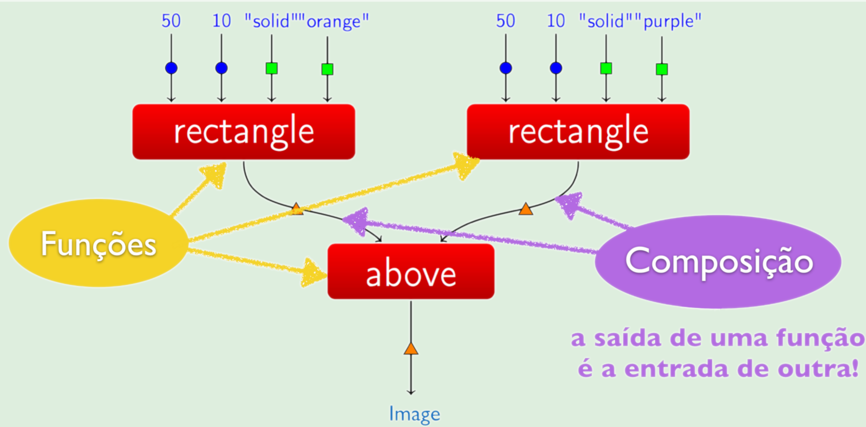
Folha com questões. Dispositivo (computador, smartphone, tablet) com acesso a internet

METODOLOGIA

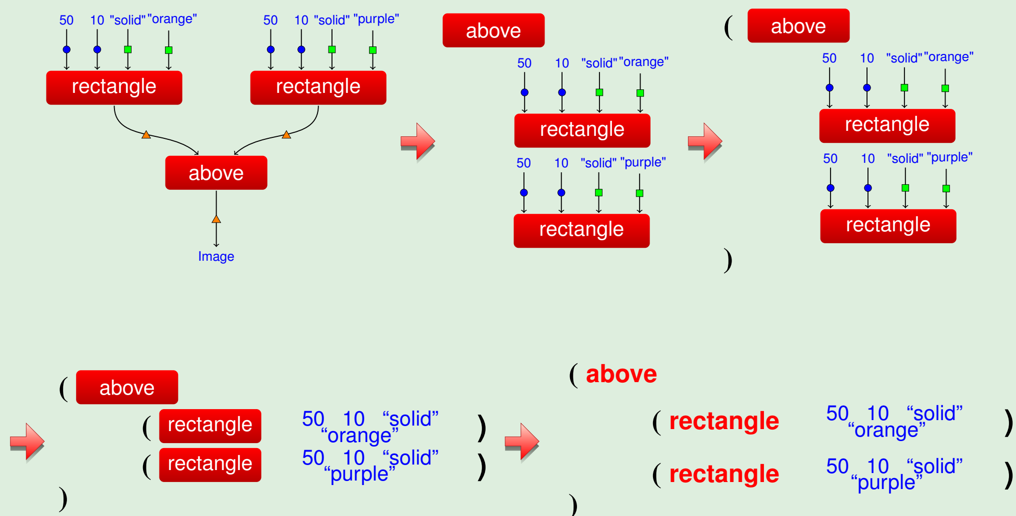
Passo 1: Diferenciar a definição de uma função de sua aplicação, apresentando uma notação gráfica para os diferentes tipos de dados.



Passo 2: Destacar a relação entre entradas e saídas de funções para construir composições bem-definidas.



Passo 3: Apresentar o processo de tradução da notação visual para a linguagem *Scheme*.



Passo 4: Avaliação

Avaliação

Solicitar a resolução dos seguintes exercícios e na sequência corrigi-los.

1. Traduza todas as bandeiras construídas na avaliação da atividade anterior para a linguagem *Scheme* e execute no *WeScheme*. *Obj.: Relacionar algoritmos descritos na linguagem visual com a linguagem textual, utilizando corretamente a sintaxe do Scheme*

2. Construa as figuras:

Obj.: Abstrair a solução visual, utilizando diretamente a linguagem Scheme e aplicar a técnica de (de)composição para construir as imagens.

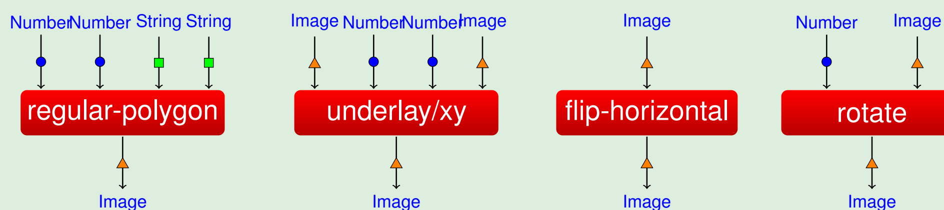
Além das funções já introduzidas nas atividades anteriores, a funções a seguir também podem ser utilizadas:

regular-polygon Dados o tamanho do lado, o número de lados, o tipo (solid ou outline) e a cor, a função gera o polígono regular correspondente.

underlay/xy Dados uma imagem $i1$, uma valor de deslocamento horizontal x, um valor de deslocamento vertical y e uma imagem $i2$, a função desloca a segunda imagem de acordo com os valores dados e coloca a imagem $i1$ sob a imagem $i2$. Inicialmente, as imagens são alinhadas no topo à esquerda.

flip-horizontal Dada uma imagem, a função gera a imagem espelhada no sentido horizontal.

rotate Dados um ângulo (em graus) e uma imagem, a função faz a rotação da imagem de acordo com o ângulo dado.



Atividade de Definição de Funções

Descrição Geral

Identificam-se partes de código a serem abstraídas, de forma a simplificar as soluções e constroem-se funções a partir da identificação de padrões.

Objetivo

Trabalhar as técnicas de abstração e generalização na definição de funções.

Habilidades trabalhadas

- Identificar que um algoritmo pode ser uma solução genérica para um conjunto de instâncias de um mesmo problema, e usar variáveis (no sentido de parâmetros) para descrever soluções genéricas;
- Identificar subproblemas comuns em problemas maiores e a possibilidade do reuso de soluções.

Materiais utilizados

Dispositivo (computador, smartphone, tablet) com acesso a internet.

Metodologia

Passo 1: Introduzir a possibilidade de simplificação de código por meio da definição de nomes.

```
(beside
  (above
    (overlay (star 10 "solid" "blue")
             (rectangle 40 30 "solid" "white"))    → RetanguloComEstrelaAzul
    (rectangle 40 30 "solid" "blue"))             → RetanguloAzul
  (above
    (rectangle 40 30 "solid" "red")               → RetanguloVermelho
    (overlay (star 10 "solid" "red")
             (rectangle 40 30 "solid" "white"))))  → RetanguloComEstrelaVermelha
```

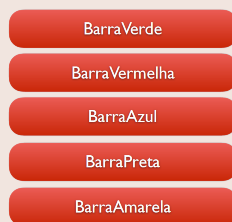
```
(define RetanguloComEstrelaAzul        (overlay (star 10 "solid" "blue")
                                                (rectangle 40 30 "solid" “white")))
(define RetanguloAzul                  (rectangle 40 30 "solid" “blue")))

(define RetanguloVermelho              (rectangle 40 30 "solid" “red"))
(define RetanguloComEstrelaVermelha    (overlay (star 10 "solid" "red")
                                                (rectangle 40 30 "solid" “white"))))
```

```
(beside
   (above RetanguloComEstrelaAzul RetanguloAzul)
   (above RetanguloVermelho RetanguloComEstrelaVermelha))
```

Passo 2: Apresentar um exemplo que permita identificar padrões nas descrições das soluções e construir uma função que generalize os padrões identificados.

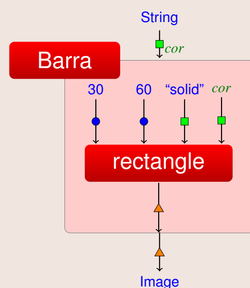
Como desenhar essas bandeiras?

Quais funções seriam úteis?



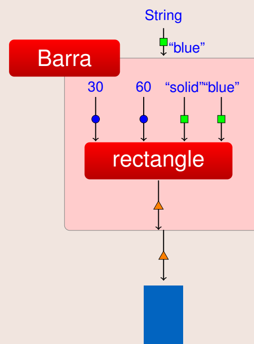

```
(rectangle 30 60 "solid" “darkgreen")
(rectangle 30 60 "solid" “red")
(rectangle 30 60 "solid" “blue")       cor
(rectangle 30 60 "solid" “black")
(rectangle 30 60 "solid" “gold")
```

Definição:




```
(define (Barra cor)
        (rectangle 30 60 “solid” cor))
```

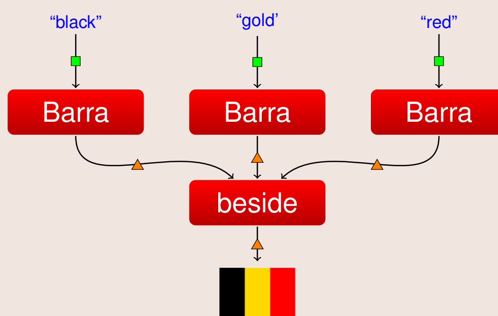
Aplicação:



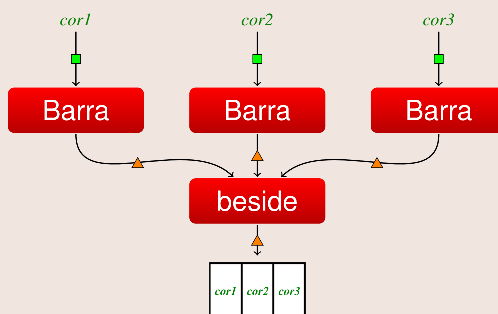
(Barra “blue”)

Passo 3: Apresentar uma metodologia para a definição de funções.

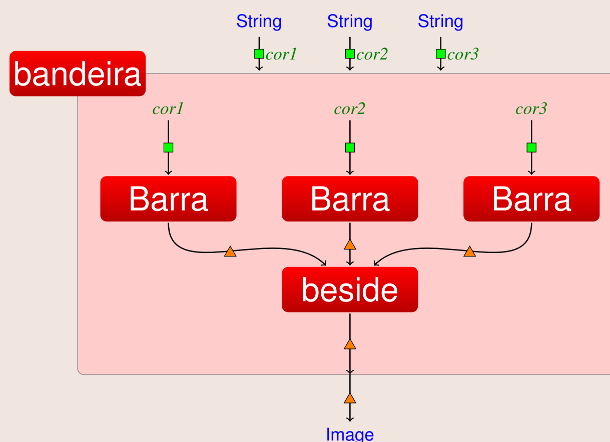
Como obter uma definição genérica de um algoritmo?



Nomear os argumentos.



Definir uma função, nomeando-a.



Definição textual:

```
(define (bandeira cor1 cor2 cor3)
        (beside
              ( Barra cor1 )
              ( Barra cor2 )
              ( Barra cor3 )
        )
)
```

Passo 4: Avaliação.

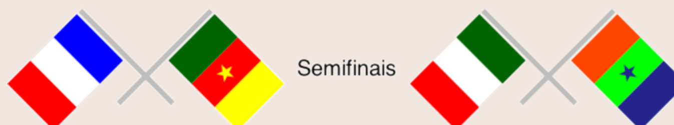
Avaliação

Solicitar a resolução dos seguintes exercícios e na sequência corrigi-los.

1. Desenhe as bandeiras, e dê nomes para elas:

2. Construa uma função para desenhar bandeiras com o seguinte formato:

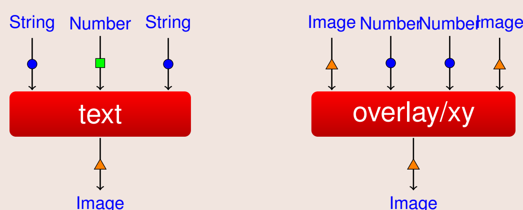
3. Construa uma função que, dadas 4 bandeiras, monta a seguinte imagem:



Além das funções já introduzidas nas atividades anteriores, a funções a seguir também podem ser utilizadas:

text Dados um string, o tamanho da fonte e a cor, a função gera uma imagem com o texto do string.

overlay/xy Dados uma imagem $i1$, uma valor de deslocamento horizontal x, um valor de deslocamento vertical y e uma imagem $i2$, a função desloca a segunda imagem de acordo com os valores dados e coloca a imagem $i1$ sobre a imagem $i2$. Inicialmente, as imagens são alinhadas no topo à esquerda.



Obj.: Identificar que existem padrões nas imagens e definir funções genéricas para construi-las.

Atividade Desenhando Cenas

Descrição Geral

Desenha-se cenas a partir de imagens, explicitando o processo de construção em si. No final, define-se uma forma de movimentação para um UFO e a cena é animada (com o UFO voando).

Objetivo

Enxergar uma cena como uma estrutura recursiva, construída através da inserção gradual de várias imagens em uma folha vazia. Também serão trabalhadas a abstração e generalização construindo funções auxiliares para montar a cena e movimentar um UFO.

Habilidades trabalhadas

- Trabalhar as técnicas de abstração e generalização na definição de funções.
- Colaborar e cooperar na proposta e execução de soluções algorítmicas utilizando decomposição e reuso no processo de solução.
- Empregar o conceito de recursão, para a compreensão mais profunda da técnica de solução através de decomposição de problemas.

Materiais utilizados

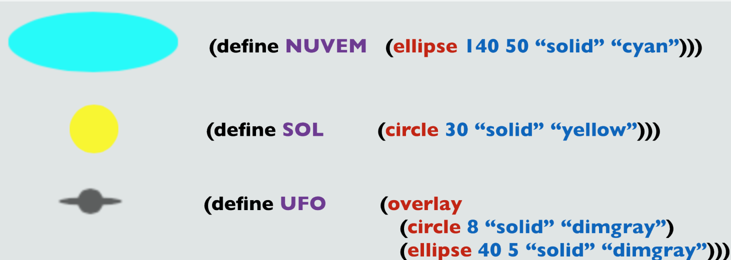
Dispositivo (computador, smartphone, tablet) com acesso a internet.

Metodologia

Passo 1: Mostrar uma CENA, identificar as partes que compõem a cena e discutir como é o processo de desenhar esta cena específica em um papel. Discutir como generalizar (desenhar qualquer cena).

Passo 2: Discutir sobre o que seria necessário para que o computador desenhasse a cena

Tarefa 1: Defina constantes para desenhar as imagens das nuvens, do sol e do UFO.



Passo 3: Identificar que as casas tem um padrão de desenho, e que sua construção pode ser generalizada por uma função.

Tarefa 2: Defina uma função que, dada uma cor de parede, desenha a casa correspondente.

```
;; casa: String → Image
;; Dada a cor da parede, desenha uma casa.
;; Exemplos:
;;    (casa “lawngreen”) =
;;    (casa "gold") =
(define (casa cor)
    (above
        (isosceles-triangle 25 120 “solid” “orangered”)
        (rectangle 40 20 “solid” cor)))
```

Passo 4: Introduzir as funções empty-scene e place-image, usadas para construir cenas, explicitando toda cena complexa nada mais é do que uma imagem inserida em uma outra cena que já contém outras imagens.

Como a gente faz um desenho?

* pega uma folha vazia — empty-scene

* coloca uma imagem na folha, em alguma posição — place-image

* coloca outra imagem na folha, em outra posição — place-image

* ...

ou seja, uma cena é construída recursivamente...

```
;; Uma Cena (Scene) pode ser
;;    1. vazia (empty-scene larg alt), onde
;;       larg : Número
;;       alt  : Número
;; ou
;;    2. (place-image img x y cena) , onde
;;       img  : Imagem
;;       x    : Número
;;       y    : Número
;;       cena: Cena
```

Tarefa 3: Defina uma constante chamada CIDADE como uma cena com algumas nuvens, um sol e algumas casas.

Tarefa 4: Para posicionar o UFO na cena, vamos calcular sua posição. Construa uma função que, dado o tempo decorrido, calcula a distância percorrida por uma nave, imaginando que ela voa a uma velocidade de 5 pixels por unidade de tempo.

Tarefa 5: Imagine que um UFO voa horizontalmente (sem variação no eixo y). Construa uma função chamada desenha-UFO que, dado o tempo decorrido, desenha a cena da CIDADE colocando um UFO na posição (x, y) desta cena, onde x corresponde à distância percorrida desde o ponto zero no eixo x (considerando a velocidade de 5 pixels por unidade de tempo), e y é alguma constante (por exemplo 200).

Tarefa 6: Na janela de interações, digite (animate desenha-UFO), clique `Run` e veja seu UFO voar!

Passo 5: Discussão e avaliação.

AVALIAÇÃO

Propor algumas perguntas para fomentar a discussão e analisar se os alunos compreenderam o conceito apresentado, por exemplo:

1. Comparem as cenas de vocês, algumas tem mais, outras menos elementos. Como essa diferença aparece na definição das constantes que representam essas cenas? *Obj: Entender que a diferença é basicamente o número de chamadas da função* place-image*, quanto mais elementos na cena, mas chamadas são necessárias.*

2. Por que a função place-image tem 4 argumentos, uma imagem, dois números e uma cena? *Obj: Identificar o papel de cada um dos argumentos da função, compreendendo que todos são necessários para que se possa realizar a ação de inserir uma imagem em uma cena.*

3. Por que a função empty-image é necessária? *Obj: Entender que toda cena é montada inserindo imagens em alguma cena já existente, então é necessário que exista uma "primeira cena", ou a cena vazia, para que o processo possa ser iniciado.*

4. Como poderíamos montar um filme a partir de cenas? *Obj: Enxergar um filme como uma lista de cenas (muito parecidas, apenas com alguns itens em lugares diferentes), identificando a necessidade da estrutura de dados lista para representá-lo.*

5. Seria possível construir um programa para montar uma cena, passando como parâmetro o número de casas que se quer na cena? *Obj: Discutir como tal programa poderia ser construído (por exemplo, qual a cor das casas? qual a posição delas na cena? como o programa executaria?), vislumbrando a ideia de montar um programa recursivo, que insere uma casa de cada vez em uma cena menor.*