Computer Science Illustrated

Ketrina Yim

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-79
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-79.html

May 21, 2009

Computer Science Illustrated:

Engaging Visual Aids for Computer Science Education

By Ketrina Yim

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report:

Committee:

Daniel D. Garcia, Lecturer SOE
Research Advisor

2009-05-20

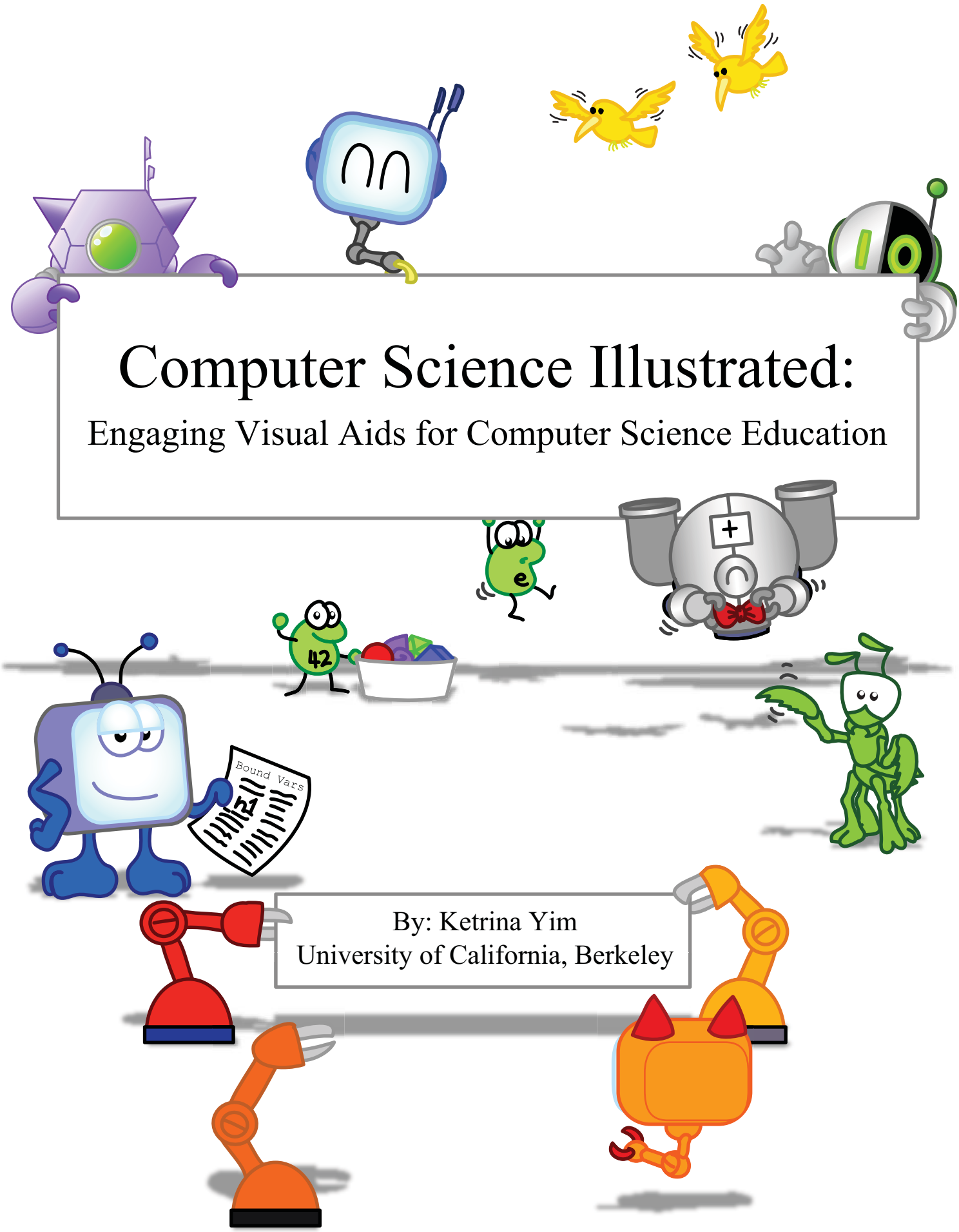
(Date)

* * * * * * *

Senior Lecturer Michael Clancy
Second Reader

5/20/2009

(Date)



Computer Science Illustrated:

Engaging Visual Aids for Computer Science Education

By: Ketrina Yim
University of California, Berkeley

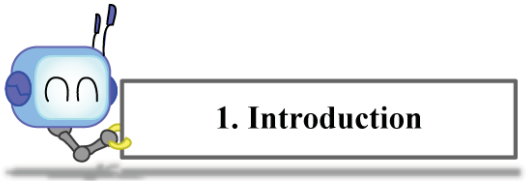
Dedication

This report is dedicated to my parents, Pony and Sue Yim, who constantly supported my academic and artistic development, and to all the teachers throughout my education who encouraged me to keep on drawing, whether it was in my notes, on my homework assignments, or on the whiteboard.

Abstract

The traditional lecture has been a standard teaching tool for university courses all around the world. However, students have different methods of learning, and for those who learn best through visual means, lectures are often not enough to understand the concepts. This is a particular problem in computer science, because a strong grasp of the fundamentals is vital to a student's understanding of the complexities of programming and computing in general.

Computer Science Illustrated is an endeavor to help visual learners comprehend computer science topics through a series of illustrations, which are made available online for use as handouts in class and posters in the computer labs. These illustrations are designed to present concepts as engaging and memorable visual metaphors combined with concise explanations or short narratives, intended to maintain the students' interest and facilitate retention. An additional goal of the project is to make learning the concepts an entertaining experience through the use of colorful and whimsical characters in the illustrations. In producing our twenty-seven illustrations, we determined which topics were most difficult for students to understand in our university's introductory computer science courses and followed a step-by-step process of design, redesign, and revision to generate resolution-independent illustrations. In this report, we will present the rationale behind illustrating computer science concepts, the systematic process we employ to create and distribute our illustrations, the challenges faced during development, and a case study detailing the creation of a specific instance of our illustrations. We will also describe the results of assessing the effectiveness of our illustrations as visual aids used in courses, and conclude with additional paths this project may take in the future.



1. Introduction

1.1 Visual Learning and Constructivism

Humans are highly visual creatures; sight is the average person's dominant method of perceiving the world. Indeed, the retinas in the eyes contain nearly 70 percent of the entire human body's sensory receptors, and optic nerve fibers comprise one-third of all the nerve fibers conveying information to the central nervous system [16]. On top of this, three of the four lobes of the human brain are involved in processing visual information received from the optic nerves for tasks such as object recognition and color processing [18]. Since sight is a human's strongest sense, it is only natural that visuals play a significant role in people's daily lives and in learning. In fact, there has been a long tradition of using images in education, spanning centuries, though empirical and systematic investigations of its effectiveness did not take place until the 1970s [19].

Strange it is, then, that the lecture has emerged as the most common teaching strategy utilized by instructors in grade schools, universities, and other educational institutions. It is characterized by the instructor giving what is best described as a lengthy oral presentation, sometimes accompanied by text and diagrams written on boards or projected on a screen, on the lesson of the day. The heavy emphasis on verbally conveying information to students makes it a primarily auditory education method, and it is this point that causes problems for many students. Education research has shown that students have preferred modes of learning, and can be categorized accordingly using the Index of Learning Styles, a set of forty-four questions designed to determine a student's learning preferences [8]. Felder defines these modes of learning using a four-dimensional space: active-reflective, sensing-intuitive, visual-verbal, and sequential-global [7]. In the visual-verbal dimension, it has been shown that students can be differentiated according to cognitive style, learning preference, spatial ability, and general achievement [14]. The students we are most interested in helping are on the visual end of the spectrum.

The relationship between a student's learning style and his or her performance in computer science courses is a topic of considerable interest in computer science education research. Chamillard's study revealed that there was a correlation between learning style and course performance, regardless of the instructor teaching the course [4]. More specifically, a study conducted by Thomas et al. showed that verbal learners performed significantly better than visual learners in introductory programming courses [20]. The implication is that current methods of teaching computer science present an advantage to students who learn verbally rather than visually. Students who learn visually learn best from images, charts, diagrams, and animations, which are often missing from computer science lectures due to the abstract nature of many concepts and the time and skill required to produce them. Thus, visual learners frequently have difficulty understanding computer science topics. The primary goal of **Computer Science Illustrated** is to assist visual learners by providing illustrations that convey the concepts. However, *all* students can benefit from this work, because it allows them to see the material presented in a fun and different way.

Constructivism also plays a role in a student's understanding of computer science subjects. According to the constructivist theory, students create knowledge structures by mentally selecting and organizing visual and verbal material and subsequently integrating it with prior knowledge they have on the subject [17]. These structures can be fallible, due to gaps in understanding. However, the gaps can go unnoticed, as they often do in computer science education, because performance does not necessarily indicate understanding [2]. As a result, some students possess mental models based on incorrect prior knowledge and require guidance to reconstruct them. At the same time, other students may enter a computer science course with no prior model of the concepts and may require guidance to prevent faulty constructions. Thus, **Computer Science Illustrated** also serves a purpose in constructivist learning by helping students establish or repair mental models.

1.2 About the Artist

As a visual learner, I often created drawings and diagrams in my class notes to reinforce my memory and help myself understand lessons. This was a strategy I carried over from middle school into my undergraduate years, and it was fueled by my passion for drawing, a trait that emerged early in my childhood. I am a primarily self-taught cartoonist, learning to draw from watching cartoons, reading comics, and sketching in my spare time. Eventually, I added graphic design to my skill set.

In my sophomore year of college, I joined the graphics team of California Engineer, a student-run magazine that published undergraduate research, and eventually became the graphics manager. As my drawings became more popular among students, I was commissioned to create t-shirt designs for several student groups, including Eta Kappa Nu, the electrical engineering honor society, and the computational game theory research and development group Gamescrafters.

Some might wonder why I did not pursue the art major, considering my interest in the subject. Indeed, I might have, had it not been for a summer science camp I took as a sophomore in high school. Not only did I learn to program in C and how to put together a computer, but I also developed a fascination with computer science. I also felt that since computer science was a subject I knew relatively little about at the time, it would be an intellectually stimulating field to explore. Thus I decided to pursue computer science, a choice that sowed the seeds of this research project.

1.3 The Origins of Computer Science Illustrated

Most research projects begin with reading papers and finding open problems; this endeavor began with cartoons. In my first computer science course, **CS3: Introduction to Symbolic Programming**, I often drew computer science-related cartoons, as well as some of the drawings I used to help grasp the concepts, on the computer lab's whiteboards. The lead teaching assistant of CS3 at the time, Clint Ryan, discovered that many students were amused and intrigued by the whiteboard drawings, and he thought they could be used as helpful reminders of the course topics. Through contact with the other teaching assistants, Clint eventually discovered I was behind the artwork, and requested that I produce some illustrations for use in lab sections. It was then the

precursor to **Computer Science Illustrated** was developed—a set of seven illustrations hand-drawn in ink depicting several aspects of Scheme, including higher-order functions, list constructors, word-sentence selectors, and empty words and sentences, and common mistakes students make when using some of Scheme's built-in functions. During my junior year, Dr. Daniel Garcia discovered these illustrations, whereupon he offered me a formal research project to produce illustrations for the benefit of computer science education. Thus it was from anonymous and spontaneous whiteboard artwork that **Computer Science Illustrated** emerged as a venture to improve computer science education.

2. Related Work

Numerous endeavors have been made to incorporate visuals into computer science education. In the commercial sphere, it is already known that attractive and understandable visuals are important to the success of an informational text. Often, the visuals in computer science texts are diagrams, graphs, or data tables, but some authors apply a more artistic touch to their work. One notable example is Larry Gonick's **Cartoon Guide to Computer Science**, a book describing the basics of computer science, such as binary numbers and logic gates, as well as the history of computing through hand-drawn diagrams and an assortment of characters [9]. While a majority of the illustrations are simply decoration or entertaining commentary for the ideas described in the text, some serve as representations or visual mnemonics of the concepts. For instance, to teach readers about how adders are used, Gonick depicts the adder in circuit diagrams as a black box with a snake, the coincidentally named adder, in it. These comical reminders help readers remember ideas by associating concrete objects with the abstract concepts. The result is a text that simultaneously reads like a graphic novel and offers a friendly introduction to basic computer science concepts.

In the space of computer science education research, attempts have been made to develop curricula that benefit different types of students, including visual learners. One such attempt is a software engineering course layout proposed by Layman that takes the students' personality types and learning styles into consideration [13]. To accommodate visual learners, Layman's course includes pictures and charts in lecture and requires students to draw diagrams for assignments and projects. Other approaches take less drastic measures, seeking to appeal to students by including multimedia in the existing curriculum instead of restructuring the entire course. Of these other approaches, most fall into the category of *algorithm visualization*.

The main goal of algorithm visualization is to facilitate the understanding of algorithms by using graphics to demonstrate how they work. Studies have shown that student engagement with algorithm visualizations increases learning [11]. Approaches can range in complexity, from simple flow charts to Biermann and Cole's "comic strip" approach, which explains binary search and splay trees through sequences of static images [3]. Attempts have also been made to present visualizations of algorithms at various levels of abstraction, such as Müldner's **Algorithm Explanation** [15]. Animation is another option considered in algorithm visualization, since seeing the algorithm in

action often helps with the understanding of its inner workings. Sorting algorithms are commonly explained through animated visualizations, as seen in Baecker's **Sorting Out Sorting**, a video explaining nine sorting methods [1]. While these visualizations have proven to be effective education tools, they are limited by three ways. First, their main purpose is to teach how algorithms work, so the methods used do not necessarily extend to all computer science concepts. Second, many algorithm visualizations, particularly animations, are presented through the computer and thus cannot be used as offline resources. Third, the visualizations rely on abstract lines and shapes, which can convey actions without visual distraction but make it difficult to form memorable mental pictures of concepts.

Our approach can be considered a complement to algorithm visualization. It covers essential concepts as well as algorithms, and the illustrations are printable, which makes them accessible away from the computer. We also make use of visual metaphors to connect computer science concepts with concrete characters and objects, which students can more easily relate to than lines, shapes, and numbered nodes. This can help students when they attempt to formulate mental images.

3.1 Development

The process of producing one of our illustrations follows the standard design process between artists and clients. Development of an illustration begins with determining which concept should be visualized. Ideally, we would produce illustrations for every concept taught in the introductory computer science courses we currently cover, but limited time and resources force us to be more selective. Topics are typically chosen from the set of subjects students most often have difficulty grasping within the courses. This information is usually gathered from discussions with the faculty teaching the courses, as they have direct experience with students' difficulties and are able to identify challenging topics from the results of various course activities. Occasionally, as in the case of MapReduce we describe in Section 3.3, the faculty will directly suggest a topic to illustrate.

Once a suitable topic has been selected, we begin planning out the illustration. At this stage, we hold meetings to discuss which aspects should be represented by images and which should be included as accompanying text. While the pictures are designed to convey as much of the information as possible, the inclusion of text is sometimes unavoidable, such as when code fragments or narrative segments form an integral part of the illustration. The product of these meetings is a set of sketches plotting out the basic structure of the illustration and rough ideas for the characters and objects that will be used as visual metaphors of the concept.

In the next step, we design the visual metaphors that will illustrate the concept. The basic ideas created in the planning stage are fleshed out in this part of the process, as we begin to consider factors such as how understandable, memorable, and aesthetically

appealing the metaphors are. To maximize effectiveness, the metaphors must be simultaneously visually attractive to engage audiences and clear and memorable to enhance retention of the represented concept. To address visual appeal and make the experience of learning from the illustrations entertaining, we design the characters in a cartoon style. We also have to take into consideration any existing metaphors that may have to work with the one currently being designed. The illustrations often contain multiple metaphors, new and old, interacting with each other, so consistency with prior metaphors is essential in preventing confusion among students. Developing the visual metaphors is both the most important and the most difficult aspect of producing an illustration.

We then take the sketched layout and visual metaphors and generate a low-fidelity prototype of the illustration. Here, the layout of the illustration is planned out, determining the location of each image and text segment. The appearance of the visual metaphors is also refined to be more aesthetically appealing and to bring it closer to the final look. The prototype is typically drawn by hand on paper, since the ability to make quick changes facilitates the process of revision.

Finally, no illustration is considered complete without revisions. Much of the revision occurs within the project group, but typographical errors, metaphor inconsistencies, and other mistakes can still slip by unnoticed in the sketches and initial prototype. It may also be possible that the prototype illustration is not as understandable as the sketches were. Thus, it is important to receive critique of the illustration from as many individuals external to the project as possible. We present the prototypes, usually as scanned images transmitted through electronic mail, to the teaching faculty and other members of the course staff and ask for feedback regarding accuracy, consistency, and clarity. Aside from corrections and suggestions for improvement, responses from the previewing audience can also include different opinions on how to present the concept and advice involving possible alternative visual metaphors. If deemed valid and incorporated into the illustration, these responses can sometimes significantly alter the appearance of the final product, as well as increase the iterations of revision needed. An illustration may undergo numerous revisions before being considered ready for distribution, but at some point it must be considered "done" so that the next topic can be tackled. For this we have a set of guidelines to determine the completeness of an illustration:

- All typographical and visual errors have been corrected
- The meaning of the text is clear
- The visual metaphors used are understandable
- For code segments, the code is free of syntax errors and the output is consistent with the input
- If the illustration contains aspects common to multiple illustrations, the visual metaphors used for those aspects are consistent

On occasion, a revision in one aspect necessitates revision in the rest of the illustration. Thus the guidelines must be consulted at every revision step. Once the illustration satisfies all of the guidelines, it is added to the collection for distribution.

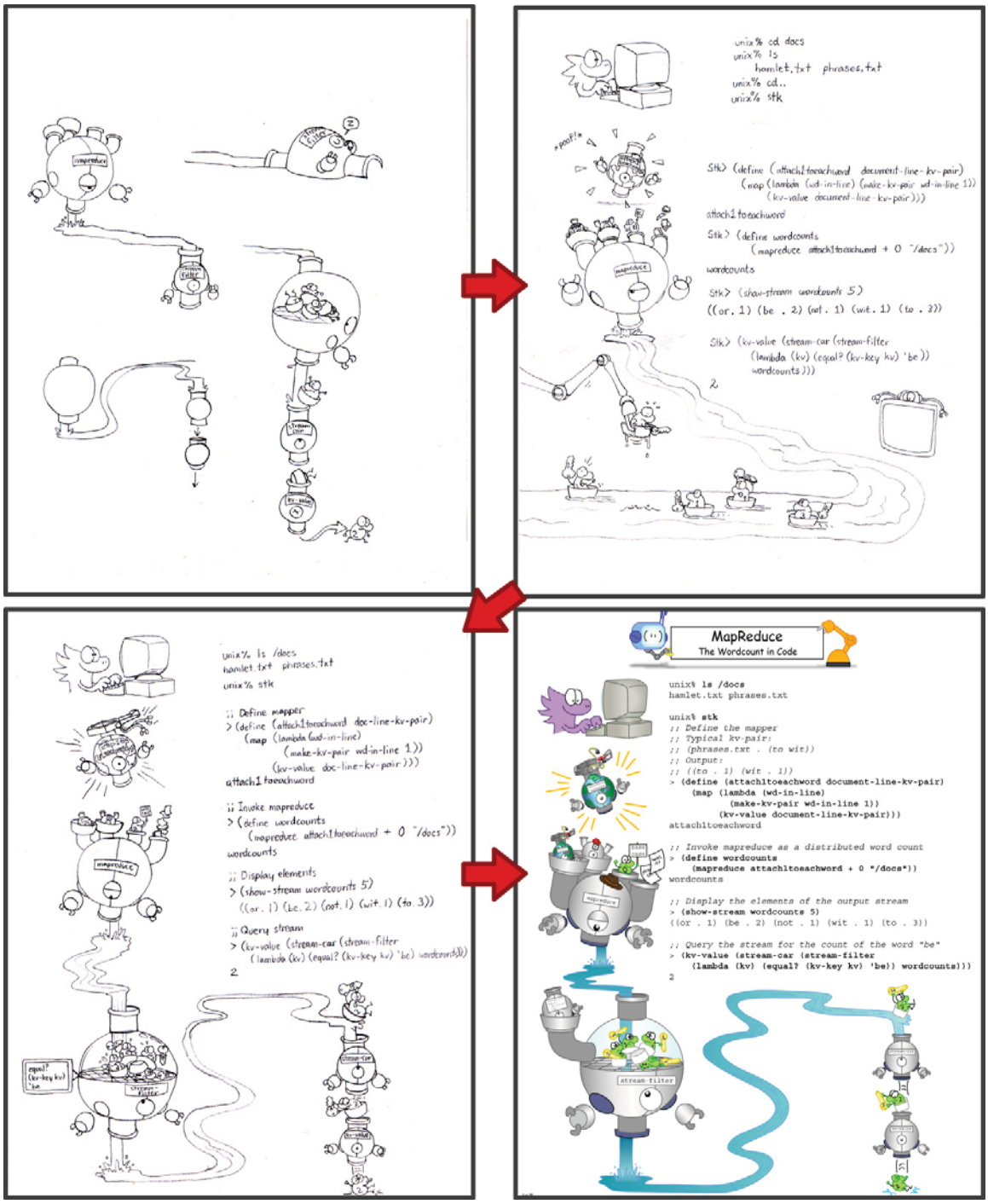


Figure 1: From sketches to final illustration: the initial sketch (top-left), the first prototype (top-right), the revised prototype (bottom-left), and the fully-colored and digital illustration (bottom-right).

Before an illustration can become part of the **Computer Science Illustrated** collection, however, it must be converted into a digital format [Fig. 1]. This means we draw the illustrations digitally. Digital drawing is an essential part of the project; it allows for online distribution of the illustrations and flexibility in printing. This step involves transforming the paper prototype into vector graphics and text in Adobe Illustrator, the industry standard for creating and editing vector illustrations. We use vectors instead of

rasterized images for the digital illustration because vector graphics are infinitely scalable without loss of quality, while scaling raster images results in image degradation and causes individual pixels to be visible. This scalability is necessary to allow the illustration to be printed in arbitrary dimensions, provided that the dimensions match the document's aspect ratio. In this stage, the prototype is scanned, imported into Illustrator, and traced over, or it is used as a reference for the digital version, which is drawn from scratch. The conversion phase is also the point at which we add color and further detail to the illustrations. The colors used tend to be bright to make the illustrations eye-catching and whimsical. Details are added to the illustrations to make them professional, and include gradients for shading, grounding shadows, and variations in character poses to prevent a "cut-and-paste" appearance. All of the work is done on drawing tablets, the standard tool of graphic designers and illustrators in industry. To make the most efficient use of time, this conversion can take place in parallel with the revision step. The illustrations are stored as Portable Document Format (PDF) files, which preserve the vectors while still being accessible to most computers.

The production process for one illustration requires two weeks, working six to ten hours per week, on average. The actual drawing and digitization of the illustrations is spread out over this span of time, but in total takes up just one to three days for the entire development. Planning and revision occupies the bulk of the two-week span. However, revision of one illustration often occurs concurrently with the planning and drawing of another illustration [Fig. 2].

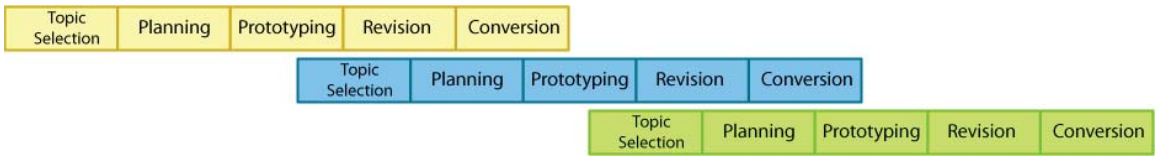


Figure 2: The development pipeline. Usually, an illustrator will work on no more than two illustrations at any given time. More illustrators mean more illustrations can be produced in parallel.

3.2 Challenges

As the description of the production process suggests, generating an illustration is by no means a simple task. There are several considerations that must be adhered to during the development process in order to ensure that the illustration is maximally effective at conveying the necessary information to students. We will describe the four challenges encountered when designing an illustration: *what* to illustrate, what *metaphors* to use, *level of detail*, and keeping *consistent*.

The first challenge usually faced occurs in the planning stage—tackling the question of *what* needs to be illustrated. As we mentioned in the previous section, we choose to illustrate what students consider to be difficult topics. The challenging concepts are not usually obtained from students directly, because many students "don't know what they don't know." In other words, many students may feel they understand a certain concept, only to find that they struggle with it during projects and exams. However, instructors and teaching assistants are often able to identify problematic areas from interaction with students during office hours and discussion, as well as from the portions of assignments and tests where students are weakest. Therefore, we gather the set of topics to illustrate from the course staff. From the collected set, we typically choose

subjects that are not usually accompanied by diagrams when presented in lecture. They are often abstract, complex, or otherwise challenging to illustrate, resulting in a scarcity of imagery that makes these topics harder for visual learners to understand. In addition, we must determine the *reasons* why students find a topic difficult to understand. For certain topics, particularly those that are specific to a programming language, the trouble could come from confusion regarding the syntax of implementations. For other topics, the difficulty may emerge from the semantics behind the concept. Identifying the problem areas allows us to decide what aspects of the topic to visualize and guides the creation of visual metaphors.

As mentioned in the previous section, designing metaphors for the illustrations is a major challenge in this project. The effectiveness of each illustration ultimately relies on the *effectiveness of the metaphors* used to convey the concepts. As Grillmeyer noted in his thesis on animations of Scheme functions, a representation that is convincing and reasonable to an expert may not necessarily help a new learner [10]. A visual metaphor can cause confusion if it is too complicated, too obscure, or is based on knowledge a novice might not have. To allow students to quickly and easily make the connection between what occurs in the illustrations and the ideas they represent, the imagery must be simple, clear, and memorable. The additional factor of aesthetic appeal enhances retention and recall of the metaphor and its corresponding idea and helps provide encouragement for students to use the illustrations as supplements to the lecture or lab. These characteristics can frequently be at odds with each other; many aesthetically appealing designs are not simple or easy to memorize, while the simplest or clearest representation may not be the most attractive. In designing a metaphor, we seek to achieve a balance among the four competing factors, a task depending heavily upon the knowledge, skill, and creativity of the illustrators. Appendix F contains a list of the metaphors that have been developed so far for the illustrations.

Often metaphor generation involves producing a design that encapsulates the most important aspects of the represented concept or element, rather than arbitrarily drawing an object or character and designating it the symbol. For instance, in Scheme lists, the characteristics to emphasize are the groupings established by the lists' parentheses and the fact that list elements are ordered. Thus lists are symbolized by ordered rows of values occupying rectangular buckets marked with parentheses [Fig. 3]. By embodying the key aspects of the concept or element, the resulting metaphors will not only serve as visual reminders of those aspects, but they will also be "appropriate." In other words, the metaphors will be understandable to students who already have a working knowledge of what they represent, allowing them to follow along when the metaphors are used to teach something they are less familiar with.

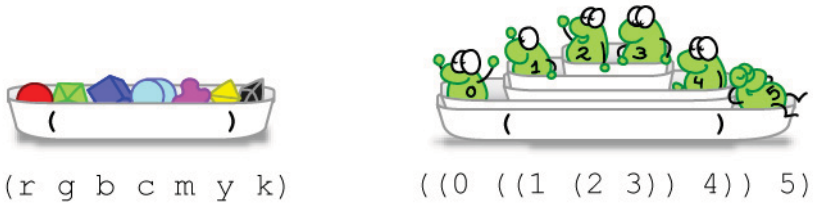


Figure 3: Visual metaphors for a simple Scheme list of arbitrary colored elements (left) and a nested Scheme list of numbers (right). The nesting of buckets makes the nesting of parentheses more apparent.

Just as clarity, memorability, understandability, and visual appeal must be balanced in each visual metaphor, compromises must be made regarding the *level of detail* provided by each illustration. Insufficient detail can contribute to confusion and reduce the illustration's usefulness, but covering too many points at once can overwhelm the viewer. Thus the depth to which a concept is covered in one illustration must be carefully considered. In general, the depth of detail in the illustrations matches the depth to which the topics are taught in the course. Providing less than this baseline makes for an insufficient supplement to lecture, while going too far beyond it could bog students down with unnecessary facts. Abstraction is a common tactic for controlling the level of detail in visualizations, but there may still be a large amount of information to convey after the abstractions are applied [10]. Additional rules are applied to manage information density in illustrations, the first of which is that concepts with multiple subtopics should be spread out over a series of illustrations. Complex subjects, such as the different types of binary integer representation, floating point numbers, or caching, fall into this category, and as such are presented as ordered sets of two or more illustrations where each covers a separate subtopic. These subtopics can be combined or further subdivided, depending on the level of detail at which these must be taught. We do try to minimize the number of illustrations per set, because a subtopic illustration and can require as much, if not more, time and resources as illustrating a full topic. The fact that some concepts require coverage with multiple illustrations brings up another development challenge—maintaining consistency.

Consistency among visual metaphors in illustrations, and among the illustrations themselves, is essential to the project. The visual metaphors are designed to help students establish a mental image of computer science concepts. Sudden changes in representations can disrupt these constructions, causing confusion and misunderstanding. Therefore, a metaphor must remain an effective representation of an aspect regardless of where it is being used or what other metaphors it is combined with in an illustration. It is also important to prevent inconsistency arising from contradictions between illustrations in a topic set or course, either through the images or in the accompanying text. There are two main cases where consistency must be ensured. First, when a concept is divided up into a set of illustrated subtopics, any visual metaphors common to the set must be compatible with the variety of ways each subtopic is depicted. Simultaneously, illustrations within the same course set can also rely on shared visual metaphors, so a visual metaphor must also be compatible with different topics within the course. For example, the representations of different Scheme functions share a common appearance, regardless of the number or type of arguments they take [Fig. 4].

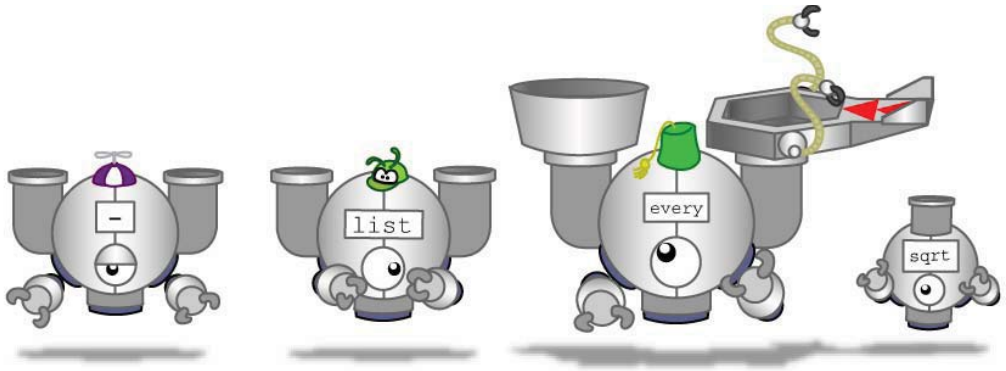


Figure 4: All Scheme functions are given the appearance of an input-output machine. The top pipes represent input, while the bottom pipe symbolizes output. In the case of `every`, a higher-order function, the input pipes feature additional components, a wide pipe for function input and a docking station for sentence input, to show that it accepts a different type of input from the other functions.

Unnecessary visual variations here could lead students to false conclusions, such as one saying functions are different datatypes. Maintaining consistency typically occurs during the metaphor design stage and is addressed by centering the design on the existing metaphors and illustrations that the new one must work with or by combining existing metaphors. Ignoring the possible relationships between the old and new metaphors is not an option, since illustrations can use multiple metaphors interacting with each other to convey information. For example, the docking station found on higher-order function machines was developed to maintain the consistency of list and sentence input as boats. If the characters' or objects' modes of interaction are not compatible, inconsistencies will emerge during the narratives.

The last challenge we will discuss mostly concerns the future of **Computer Science Illustrated**, though it is also relevant to the present. An additional objective for us is to make the project multi-generational, having it continue long after the original illustrators have graduated. Of course, this continuation would involve recruiting new students, who will inevitably have different thoughts, beliefs, and artistic styles. Currently, the issue is minor for now since there are just two artists, but it can only grow as more illustrators are involved. Differing mindsets and styles of drawing have implications in metaphor design and consistency. It is possible for each illustrator to have a different idea for visualizing a certain concept. At the same time, variance in drawing styles can create visual incongruence in an illustration or set of illustrations, even when the same metaphor is drawn. However, forcing a standardized style can stifle the expressive capabilities of the illustrators and possibly discourage potential artists from joining or staying in the project. Therefore, we have considered several measures to maintain visual consistency and allow for artistic differences. One is to have each illustrator focus on a particular subject. While subjects are not always fully isolated from each other, grouping by subject eliminates inconsistencies that would occur if multiple artists were to work on the same illustrations. Another is to unify illustration sets through the use of a standardized color palette. The idea here is that though the visual metaphors may vary in appearance due to stylistic differences, common colors will act as an indication that two depictions of a metaphor are not two separate metaphors.

3.3 Case Study: MapReduce

In this section we describe a specific instance of developing illustrations, for a Scheme-based implementation of MapReduce [6]. Originally the product of Google, it was introduced in 2008 to one of Berkeley's Scheme-based introductory computer science courses, **CS61A: Structure and Interpretation of Computer Programs**, by Brian Harvey in an effort incorporate cluster computing fundamentals into the early stages of the students' education [12]. Later on, it also became a component of another introductory computer science course at Berkeley, **CS3: Introduction to Symbolic Programming**, also taught in Scheme. The novelty of the MapReduce implementation, combined with the fact that Scheme was a language rarely used by other campuses, meant that there was a shortage of outside resources, particularly visual material, for students to consult when faced with the MapReduce curriculum. We determined that Scheme-based MapReduce was an ideal candidate for illustration.

One of the first things we encountered was the complexity underlying the Scheme-based MapReduce. CS61A is a course that teaches the Scheme list, pair, and stream datatypes. Thus, the implementation of MapReduce takes a mapper, reducer, a data value serving as the reduction's base case, and the file directory as input and produces an output stream of key-value pairs that must be queried with stream accessing functions to extract results. We decided that there would need to be an *introduction* to the MapReduce function, detailing its input arguments; a description of the processes taking place *within* a MapReduce function call; and example code depicting a *usage* of MapReduce and its resulting output. We divided these up into multiple illustrations: one to provide the basic overview, one to visually depict the inner workings, and one to present an example interpreter sequence demonstrating MapReduce and the stream accessing functions in action.

Additionally, due to curricular differences, MapReduce is taught differently in CS3 and CS61A. In CS3, where students are not exposed to Scheme's stream datatype, MapReduce is presented as three functions: `reduce-map-letter`, `reduce-map-word`, and `reduce-map-sentence`. For input, these functions take a mapping function, a reducing function, and the directory of files to be processed, and as output they produce a single element. Since the output is one value, emphasis is placed on the reduction step, where the order in which the data is processed can affect the outcome of MapReduce. On the other hand, in CS61A MapReduce is presented as a single `mapreduce` function that takes a mapper, reducer, the reducer's base case, and the file directory and outputs a data stream. The internal workings of each course's implementations also differ significantly. As a result, we decided it was necessary to create separate sets of illustrations for each course. Both CS3 and CS61A would have the overview and visualization of the internals, but CS3 would have additional illustrations to explain the differences among the three MapReduce functions and how the reduction step can create different outputs for the same input.

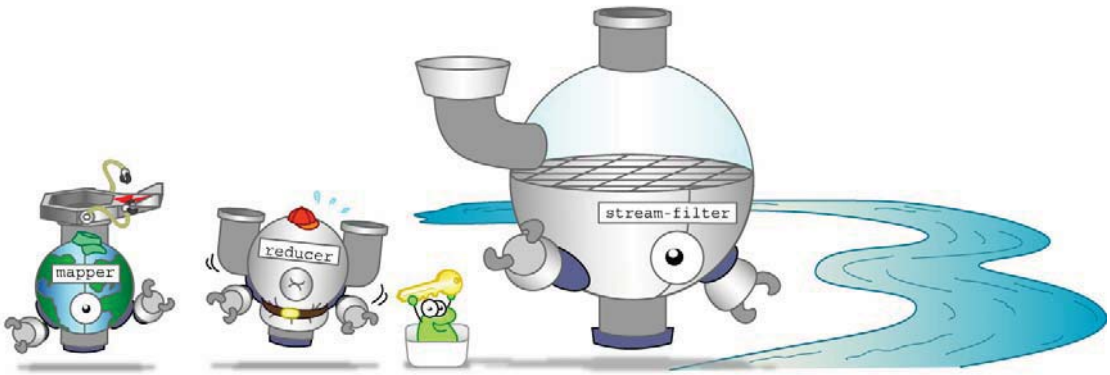


Figure 5: Metaphors used in the MapReduce illustrations. From left to right: the mapper and reducer as function machines, a key-value pair as a tub containing a value holding a key, stream-filter as a heavily modified function machine, and the output stream as a literal stream of water.

As is the case for most of our illustrations, the MapReduce processes were presented as pictorial narratives, so that students would watch the process unfold by following the list element characters as MapReduce and its helper functions process them [Fig. 5]. To keep consistent with the existing Scheme illustrations, the MapReduce functions were represented by input-output machines and lists and pairs were shown as tubs of ordered list elements. However, some modifications were made to several functions and the representation of pairs to act as visual mnemonics of their purpose and highlight certain aspects. Some modifications were applied to elements shared by both sets of illustrations. The mapper function was colored with a globe pattern, as opposed to the standard silver, to remind students that such functions were meant to "map" lists and sentences. The reducer was depicted as a machine with a belt squeezing its midsection, to emphasize the fact that it compresses lists to single elements. Other modifications were applicable only to the CS61A MapReduce. Key-value pairs were still depicted as tubs, but the values within them held large metal keys labeled with the pairs' word keys. Finally, `stream-filter`, one of the stream accessing functions, was illustrated as a function machine with a transparent top half revealing the filter inside. We also created one completely new visual metaphor for the CS61A illustrations—a stream of water with pairs floating on it for the output stream.

Revisions and changes in the example code occurred throughout development, and had significant impact on the illustrations' appearance. For instance, the interpreter sequence illustration initially contained only the definition of the mapper function and the MapReduce call. When it was decided that it would also be necessary to include data extraction from the output stream, the interpreter sequence changed to include `stream-filter` and other stream access functions. In turn, this created the need to develop metaphors for those functions and incorporate them into the illustration [Fig. 6].

The results of this development were seven MapReduce illustrations: three for CS61A, and four for CS3. These illustrations, along with descriptions of their respective courses, are presented in Appendices B and C. They were distributed to the students of the courses as handouts in lecture, one of the several routes that our illustrations travel after development.

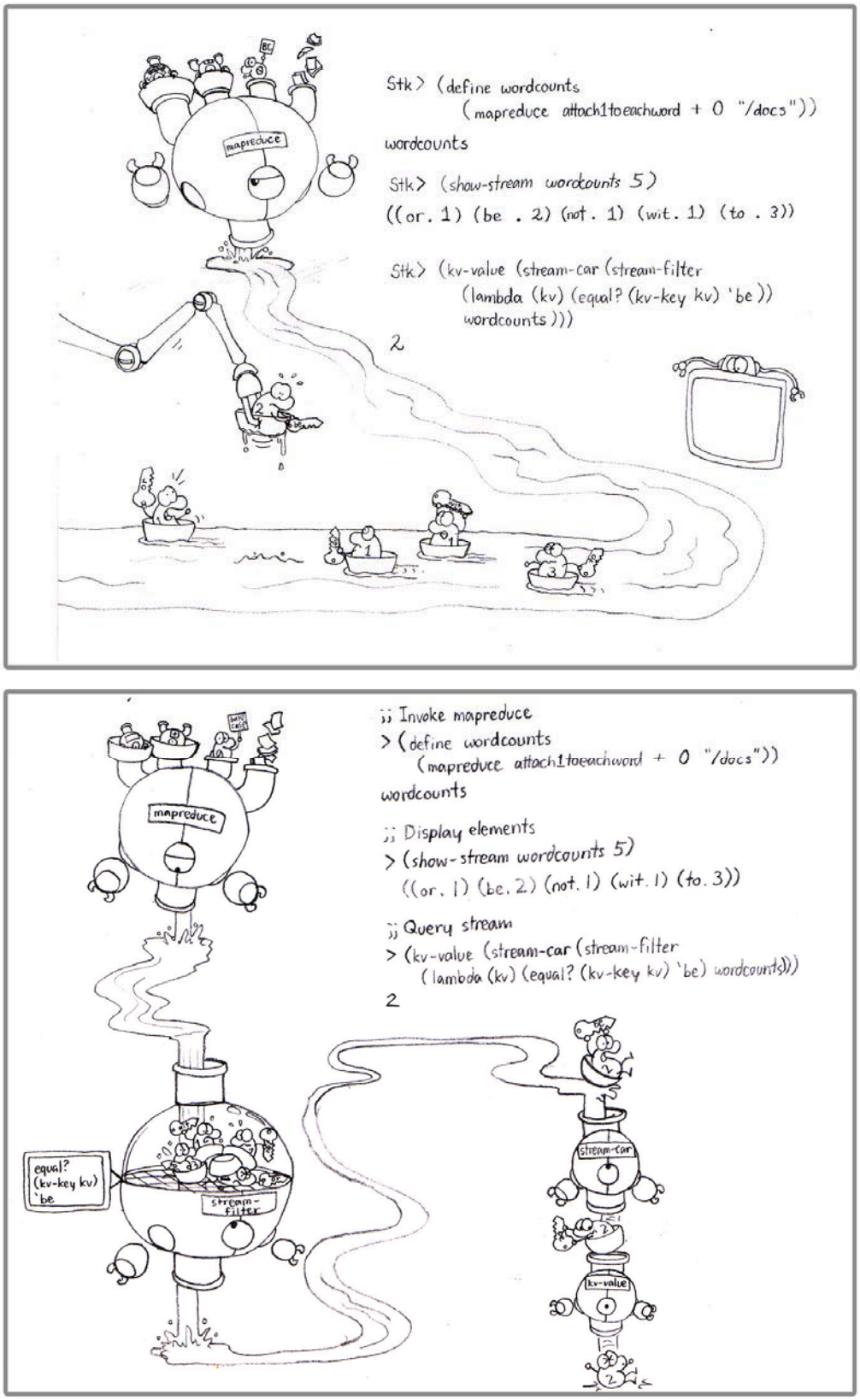


Figure 6: Two sketches of the Scheme MapReduce interpreter sequence. Initially, the illustration focused on the fact that MapReduce produced an output stream (top). Later, to explicitly show how results are extracted from the stream, stream-accessing functions were included in the illustration.

4. Distribution

Once an illustration is produced, there are three main methods that it can be put to use in a computer science course. Online distribution is one avenue we have established. The illustrations that we have created so far are available in PDF on a publicly accessible website, along with brief descriptions of the project, the illustrators, and recurring characters in the illustrations. The PDF files can be viewed in the web browser, if the user has a PDF reader plugin, or downloaded for offline viewing and printing.

The second use we intend for the illustrations is as course handouts, supplementing the lecture or discussion sections. Handouts are not a common feature in computer science courses, as they can be time-consuming to produce, but they can be invaluable to visual learners as a reference. These handouts can be distributed during class and used as a starting point for presenting topics, or provided afterwards as summaries of the lessons covered. We foresaw the possibility of printing illustrations as handouts, so all those currently produced are designed to fit the standard 8.5”x11” paper size. The illustration PDF files are also ready to print as soon as they are opened in a PDF reading program.

Finally, the resolution-independence of our graphics also allows larger scale printing, allowing users to print them as computer lab posters. Given the fact that our illustrations are currently structured in a portrait, rather than landscape, layout for course handouts, our initial prints were 2’x3’ posters. These can serve as semester-long reminders of the concepts while students work in the computer labs or as a quick reference. Due to the different aspect ratios between the handouts and posters, two versions of each illustration are created, one of which has slight adjustments to accommodate the poster dimensions. Our posters are printed by BigPosters.com on glossy paper with archival ink to protect them from fading under frequent light exposure. For additional protection, we also frame the posters prior to placing them on the computer lab walls. We use thin black poster frames, which minimize visual distraction away from the poster, purchased from Aaron Brothers. Each poster takes approximately ten days to process, print, and ship, and costs $56.25 to produce, $31.25 for the poster printing and $25.00 for the frame.

With these distribution options, particularly online availability, a couple issues need to be addressed. One is the matter of intellectual property. Since the illustrations are meant to be freely distributed, steps must be taken to prevent them from being used for profit. We considered two possibilities: copyright and Creative Commons licensing [5]. Copyright laws are automatically applied to online content, but they impose restrictions on distribution by requiring every user to ask for permission to use the illustrations. A Creative Commons license, on the other hand, allows the waiving of certain rights to facilitate free distribution.

Figure 7: The Creative Commons license chosen for Computer Science Illustrated. It is placed at the bottom of the homepage and illustrations page on the website.

Thus, we opted for the Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License to enable non-commercial use of the illustrations and the creation of derivative works based on them [Fig. 7]. The other issue is the illustrations' applicability outside of UC Berkeley. Currently, nearly half of the illustrations created cover topics involving Scheme, as they cater to CS3 and CS61A. However, Scheme is used in very few other schools, so illustrations of this category would have limited usefulness. Therefore, we have moved on to focus more on the topics of non-Scheme courses, such as CS61C: Machine Structures. Further details regarding this expansion are described in the future work section. To appeal to campuses other than Berkeley, we structured the website to organize the illustrations by topic rather than by course name.

Informal presentations of **Computer Science Illustrated** have shown that the reception of our project from students and faculty is overwhelmingly positive. However, we realize the need to quantitatively measure the illustrations' effectiveness. Our illustrations are currently designed for Berkeley's introductory computer science courses, so we held experimental assessments on the students taking them. We conducted two such assessments so far, one in CS3 and one in CS61CL, a lab-based version of CS61C. We chose these two courses because their lab sections are presented through UCWISE, a platform for displaying lessons, activities, and student assessments online [21].

For the study conducted on CS3, which took place early in the semester, the lab sections of the course were randomly divided into two groups: the control group and experimental group. The students were not informed of this division, and all received the same instruction in lecture. During the lab sessions, students were given their regular short quizzes to test their understanding of the concepts covered. However, the experimental group was allowed to view three illustrations relevant to the topics covered so far, in this case the word and sentence datatypes and functions involving them, before taking the quiz. The quiz consisted of seven questions concerning the word and sentence datatypes. To ensure fairness, in case the illustrations offered a significant advantage, every student received the illustrations at some point; the control group was allowed to view them after completing the quiz. The quiz scores collected were then analyzed to gauge whether the illustrations were helpful for the students' understanding of the concepts.

This method of assessment required careful consideration of the types of questions we wanted to include in the quizzes. Questions that specifically asked for details explained by the illustrations or the lesson text would give an unfair advantage to the experimental and control group respectively, which would potentially create

misleading results. Thus questions did not involve examples found directly in either the illustrations or the text. Moreover, we considered the timing with which the quizzes were administered. Providing the illustrations to the experimental group minutes before the quizzes most likely would not allow time for students to fully comprehend the metaphors and remember the ideas.

On that note, we decided to structure the second study, conducted before the second round of midterms on CS61CL, a little differently. The procedure was very similar to that of the first study, except the illustrations, covering caching, were presented the week prior to the quiz [Fig. 8]. This gave the students of the experimental group more time to study the illustrations and us the chance to determine if any deep learning was taking place. Additionally, the control group was given text-only equivalents of the illustrations, so that the assessment would be a direct comparison between illustrated lessons and text-based lessons. The quiz consisted of three questions about caching and cache misses in general. Again to maintain fairness, the control group received the illustrations and the experimental group received the text equivalents after the quiz.

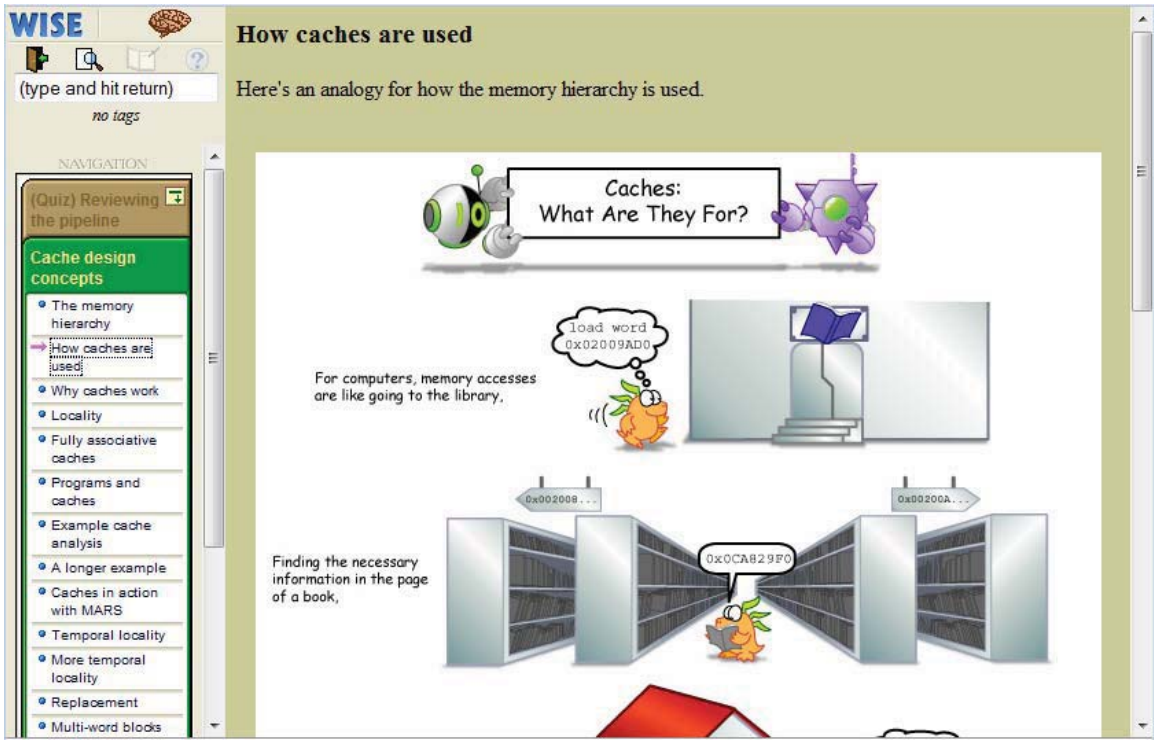


Figure 8: An illustration incorporated into a UCWISE activity for CS61CL: Machine Structures.

The study conducted on the CS3 students suggested that student understanding of concepts improved when the lessons were supplemented with the illustrations. On average, the group of students who received the illustrations scored higher on the quiz than the control group. The average score experimental group was 6.053, while the control group had an average score of 4.771. Both scores were out of seven points.

On the other hand, the study conducted on the CS61C students produced inconclusive results. There was no significant difference between the average quiz scores of the control and experimental groups, which were 2.429 and 2.433 respectively. These quiz scores were out of three points. Several conclusions can be drawn from this

outcome. It may be that the quiz was too short or the questions too general to produce responses indicative of the students' understanding of the concepts. Perhaps the supplements provided improved comprehension, regardless of whether they were purely text or illustrations. It is also possible that in giving both groups a week to absorb the information, the students consulted other resources and achieved nearly equal levels of understanding. Future assessments will require longer quizzes and stronger consideration of the types of questions to ask.

In addition to conducting the experimental assessments, we also surveyed the CS3 students for their opinions of the illustrations. We added four questions pertaining to the illustrations to the course survey that was given to the students at the end of the semester. The first asked students to rate the effectiveness of the illustrations with a score from one to seven, with seven being the highest score; the second asked if illustrations should be presented for other computer science courses; the third had students select all the ways they used the illustrations; and the fourth was an open-ended question asking for overall comments on the illustrations. Figure 9 contains tables of the results from the first three questions.

The survey revealed that the students' reception of the illustrations was very positive in general. The average rating students gave the illustrations was 5.12, with the most frequent rating being a 6. More than half the students wanted to see illustrations offered in other computer science courses, and most students used the illustrations as introductions and to create mental models of the concepts. In the open-ended responses, most of the students who used the illustrations stated that they were "helpful in visualizing what some of the functions did," "easy to read and understand," "a cool way to teach CS to beginners," "amusing", and "cute." Some said that the concepts were "easier to comprehend" after seeing the illustrations. A couple of students also claimed to be visual learners, and thought the illustrations were very useful to them as a result. Among the small minority of students who did not find the illustrations helpful, the most common comment was that the textbook and other course material were adequate for their understanding, suggesting these students already had a good grasp of the concepts prior to viewing the illustrations.

One important discovery made during the studies was that the results of assessments, whether they were quizzes or informal discussion, offered valuable insight for revisions, as the students' responses could reveal weaknesses in the illustrations. For instance, when the illustration covering Scheme list constructors was presented during the CS3 lecture, the instructor found that some students took the visual metaphors too literally and mistakenly assumed the `list` and `append` functions took exactly two arguments, instead of arbitrarily many arguments. This problem emerged from the fact that the examples given by the illustrations used only two arguments. We took this issue into consideration, and made plans to create a new illustration mapping out the effects of several numbers of inputs on the list constructor functions. As the illustrations become more widely used, we can expect more of this type of feedback emerging from their application in courses, which we will undoubtedly use to further improve the quality of the illustrations.

How effective was CS Illustrated in helping you understand the material?

Score	Number of Students
7 (very effective)	11
6	21
5	19
4	13
3	1
2	1
1 (not effective)	4

Would you like to see more or less of CS Illustrated images in other computing classes?

Score	Number of Students
A lot more	23
A little more	24
About the same	18
A little less	3
A lot less	2

How did you use CS Illustrated?

Usage	Number of Responses
As an introduction	37
As a summary	28
For remembering the "big idea" of a concept	32
To help form a mental model	37
Only read them as part of lecture notes	1
Didn't use them	2

Figure 9: Results from the survey questions given to CS3 students at the end of the semester. 70 students submitted responses. For the third question (bottom), multiple answers were permitted in case students had more than one use for the illustrations.

There are multiple routes that **Computer Science Illustrated** could follow in the future. One of the most logical is to continue expanding and revising the collection of illustrations for the introductory computer science courses. Part of this would involve finding more topics to illustrate in the courses currently covered and further improving the illustrations already in the collection. Simply because an illustration satisfies the guidelines we described in this paper does not mean there is no room for improvement. Expanding the collection also involves extension to cover topics in data structures, taught in an introductory course known as CS61B in Berkeley, which were set aside in focusing

on topics regarding Scheme and machine structures. This expansion of the collection will also demand continued assessment of effectiveness, as insights gathered from the data will help guide the project's development and improve the illustrations.

Another branch of future work could be to illustrate enough topics in a course such that the course's entire curriculum, or at least a majority of it, could be taught through the products of **Computer Science Illustrated**. This endeavor would most likely involve using illustrations as handouts, producing illustrated lecture slides, and developing lab or discussion activities based on the illustrations and the narratives within. Here the illustrations would play a much more active role in the course, rather than being passively presented to the students as course supplements. This presents the opportunity to create more engaging computer science courses through active learning and refreshed representations of course concepts.

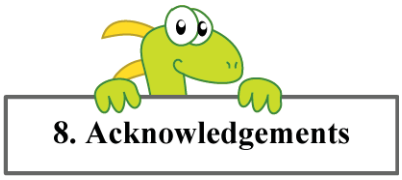
A third future work option is to extend the project's coverage to include advanced computer science topics, such as those taught in upper-division courses. Advanced topics, such as theory, artificial intelligence, and databases, tend to be more abstract, complicated, and difficult to visualize than subjects presented in the introductory courses. Even those often accompanied by many images, such as computer graphics, can be challenging for students to understand. Potentially, visual learners could benefit even more from illustrations in advanced courses due to the increased problems they might have in creating mental images.

Yet another route we could take in **Computer Science Illustrated** is to increase awareness of the project to expand its usage. Currently, despite being available online, use of the illustrations is largely limited to within the introductory computer science courses of Berkeley. Of course, this restriction is primarily due to the fact that most of the illustrations are quite specific to Berkeley, particularly the ones covering Scheme. This extension of the project would be best pursued after the collection is further expanded to cover data structures or advanced computer science subjects. The intention would be to encourage other schools to use **Computer Science Illustrated** in their courses, possibly even to create similar ventures to unite art and computer science for the benefit of education.

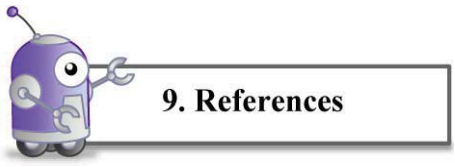
The most ambitious extension of **Computer Science Illustrated** would be to go beyond the space of static images and produce animations to convey the concepts. Given its popularity in both film and television, computer-generated, three-dimensional animation may be the medium of choice to maximize engagement of students. It would also be an ideal opportunity to incorporate techniques from animated algorithm visualization. For example, the boat metaphor could be combined with sorting animations to explain list sorting in Scheme. The tradeoff is that this type of animation requires a large time and resource investment, which could be mitigated with a sufficiently large group of artists. A less intensive alternative is to utilize digital two-dimensional animation, such as Flash animation, to more closely adhere to the current style of the illustrations. Additionally, interactivity could be included in these animations, creating small activities or games, to encourage further engagement by allowing students to actively participate. Regardless of the medium chosen to animate, both would require significantly more time to produce than static illustrations, especially with interaction involved. As such, topics to animate would have to be chosen even more carefully.

We have presented an approach to facilitate computer science education through the use of visually engaging and informative illustrations made available to students as course handouts, an online resource, and large posters to view in the computer lab. The detailed process we go through and the challenges we must address for every illustration ensures the visual metaphors encapsulating the concepts are understandable, memorable, and consistent within the groups they are used. The products of our efforts were sixteen illustrations for CS3 and CS61A and ten illustrations for CS61C, all available online and included in Appendices D and E.

The assessments we conducted to evaluate the effectiveness of the illustrations suggest that they are useful as supplements to the lessons taught in class. Surveys have also shown that students are highly in favor of using the illustrations as entertaining introductions to the concepts, as well as guides to forming mental models. Student feedback has also been useful in revealing areas in which the illustrations could be improved. With further assessment, revision, and expansion of the collection, **Computer Science Illustrated** could be a component of every computer science course at Berkeley.



First, I would like to thank Dan Garcia, my research advisor, and second reader Michael Clancy for supporting the project, providing assistance during the assessments, and taking the time to read and revise this report. I also want to thank Nate Titterton and Colleen Lewis for providing access to UCWISE and teaching me how to use the system to conduct the assessments. Additionally, I would like to thank the Weiner fund and the Office of Educational Development, who provided Instructional Improvement Grant, for financially supporting the project. Finally, I wish to thank Sally Ahn for lending her wonderful creativity and artistic skill to the project and for making it possible for **Computer Science Illustrated** to continue beyond my graduation.



[1] Baecker, R., *Sorting Out Sorting*. Videotape, 1981.

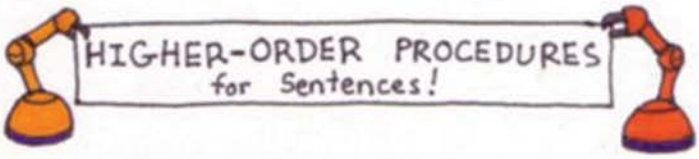
[2] Ben-Ari, M., Constructivism in Computer Science Education. Proceedings of the 29th SIGCSE technical symposium on Computer Science Education; 1998 Feb 25-28; Atlanta, GA. New York: ACM Press; (c1998) 257-261.

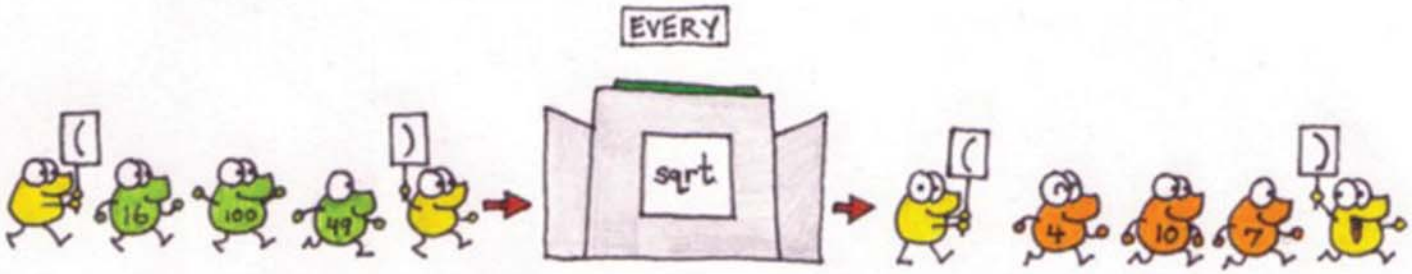
[3] Biermann, H. & Cole, R., *Comic Strips for Algorithm Visualization*. NYU Technical Report 1999-778, (1999).
[4] Chamillard, A. & Karolick, D. Using Learning Style Data in an Introductory Computer Science Course. *ACM SIGCSE Bulletin*, Vol. 31 No. 1 (Mar 1999) 291-295.
[5] Creative Commons. Creative Commons. Website. http://creativecommons.org/
[6] Dean, Jeffery & Ghemawat, S., MapReduce: Simplified Data Processing on Large Clusters. Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implementation; 2004 Dec 6-8; San Francisco, CA. Berkeley: USENIX Association; (c2004) 10-10.
[7] Felder, R., Learning and Teaching Styles in Engineering Education. *Engineering Education*, Vol. 78 No.7, (1988) 674–681.
[8] Felder, R. & Spurlin. J., Applications, Reliability, and Validity of the Index of Learning Styles. *International Journal of Engineering Education*, Vol. 21 No. 1 (2005) 103-112.
[9] Gonick, L., *The Cartoon Guide to Computer Science*. New York: Harper & Row, Publishers, Inc., (1983).
[10] Grillmeyer, O. Animations of Scheme Functions and Algorithms. Website. http://oliver.grillmeyer.googlepages.com/thesis. (Fall 2001)
[11] Grissom, S. et al., Algorithm Visualization in CS Education: Comparing Levels of Student Engagement. Proceedings of the 2003 ACM symposium on Software Visualization; 2003 Jun 11-13; San Diego, CA. New York: ACM Press; (c2003).
[12] Johnson, M. et al., *Infusing Parallelism into Introductory Computer Science Curriculum using MapReduce*. University of California, Berkeley Technical Report UCB/EECS-2008-34, (2008).
[13] Layman, L. et al., Personality Types, Learning Styles, and an Agile Approach to Software Engineering Education. *ACM SIGCSE Bulletin*, Vol. 38 No. 1 (Mar 2006) 428-432.
[14] Mayer, R. & Massa, L., Three Facets of Visual and Verbal Learners: Cognitive Ability, Cognitive Style, and Learning Preference. *Journal of Educational Psychology*, Vol. 95 No. 4, (Dec 2003) 833-846.
[15] Müldner, T. & Shakshuki, E., A New Approach to Learning Algorithms. Proceedings of the International Conference on Information Technology: Coding and Computing; 2004 Apr 5-7; Las Vegas, NV. Los Alamitos: IEEE Computer Society; (c2004).
[16] Noback, C. and R. Demarest, *The Human Nervous System: Basic Principles of Neurobiology*. McGraw-Hill, Inc., (1981).
[17] Otero, J. et al., *The Psychology of Science Text Comprehension*. Mahwah: L.Erlbaum Associates, (2002).
[18] Rolls, E. and G. Deco, *Computational Neuroscience of Vision*. USA: Oxford University Press, (2002).
[19] Schnotz. W., Towards an Integrated View of Learning From Text and Visual Displays. *Educational Psychology Review*, Vol. 14 No. 1 (March 2002) 101-120.

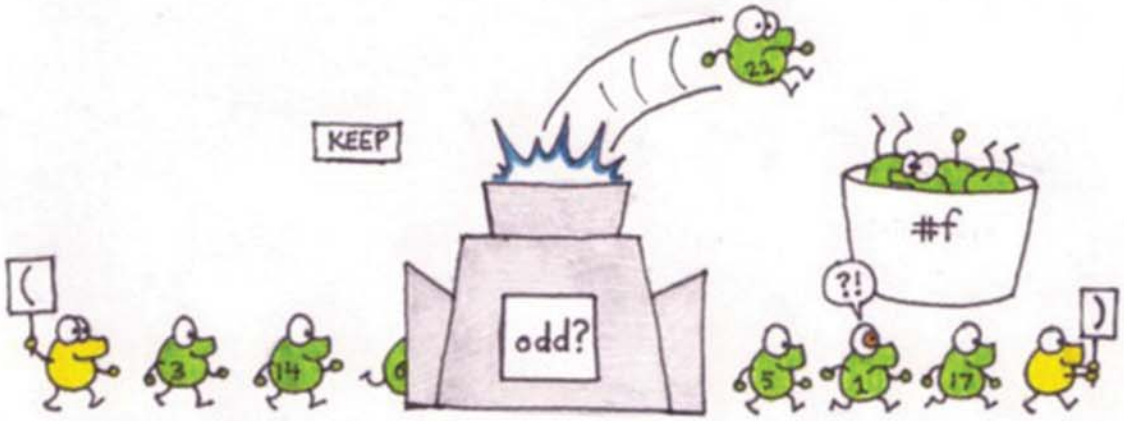
[20] Thomas, L. et al. Learning Styles and Performance in the Introductory Programming Sequence. *Inroads*, Vol. 34 No. 1, (Mar 2002) 33-37.
[21] UCWISE. UCWISE. Website. http://www.ucwise.org/.

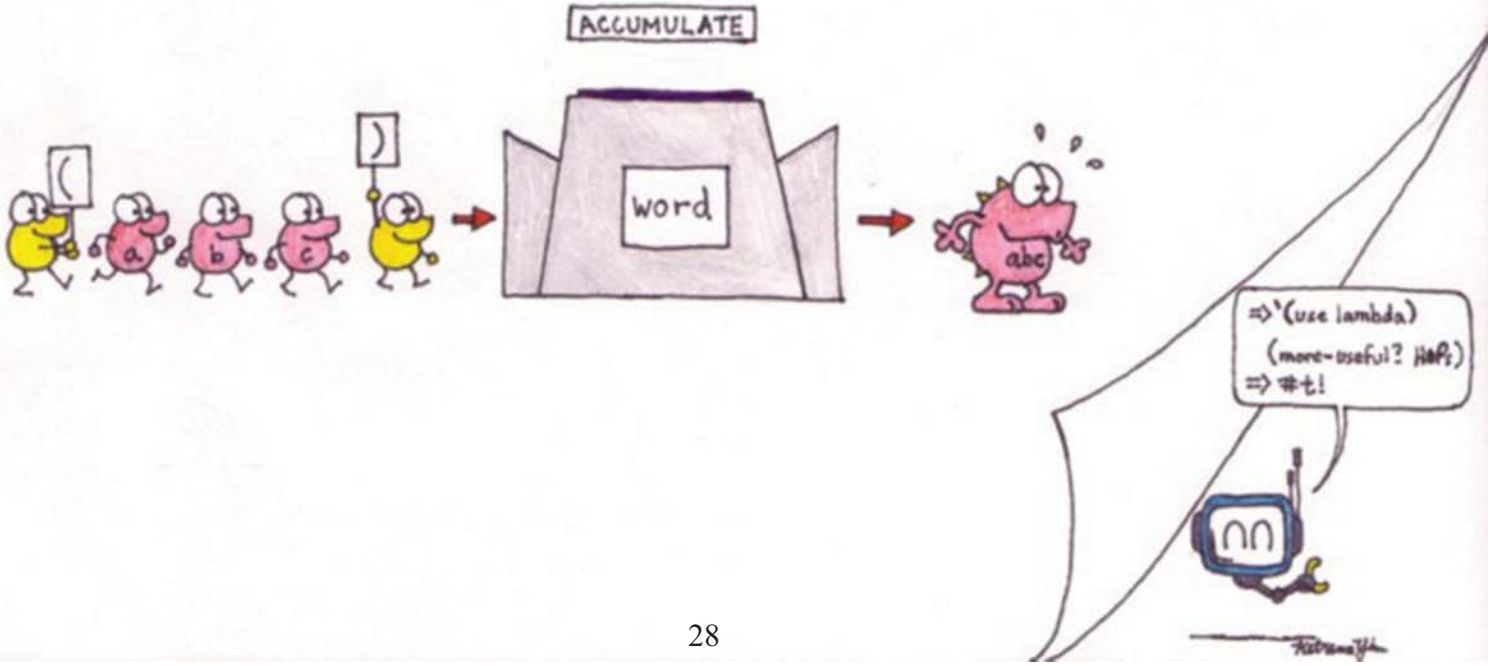
10.1 Appendix A: The Precursors to Computer Science Illustrated

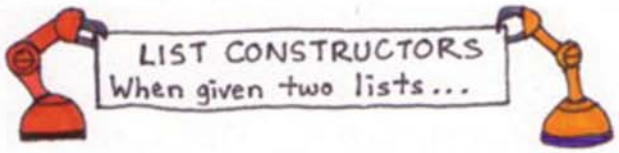
The following are the hand-drawn illustrations that were created prior to the formation of the project. These illustrations went through a less rigorous process of development and revision than their digital counterparts, but they laid the foundations for the project. The illustrations are presented in the following order:


HIGHER-ORDER PROCEDURES
for Sentences!


EVERY
(
16
100
49
)
sqrt
(
4
10
7
)


KEEP
22
#f
?!
(
3
14
odd?
5
1
17
)

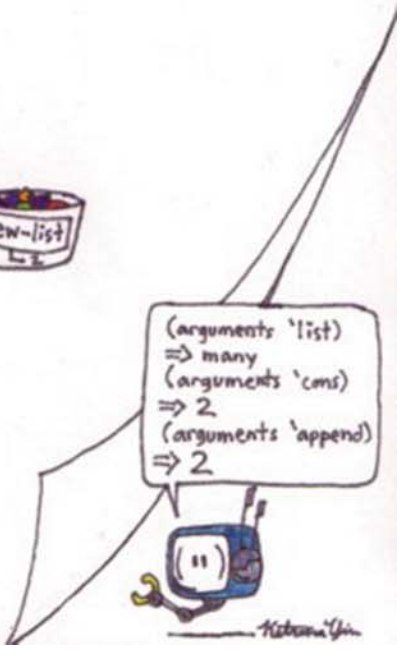

ACCUMULATE
(
a
b
c
)
word
abc
=>'(use lambda)
(more-useful? HOF;)
=> #t!

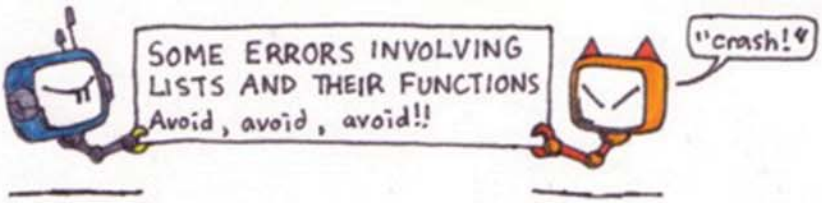


LIST

CONS

APPEND

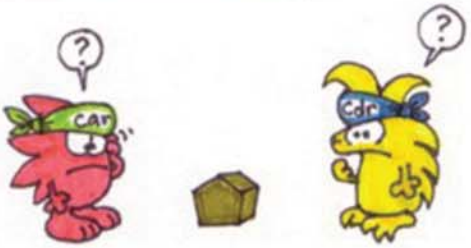




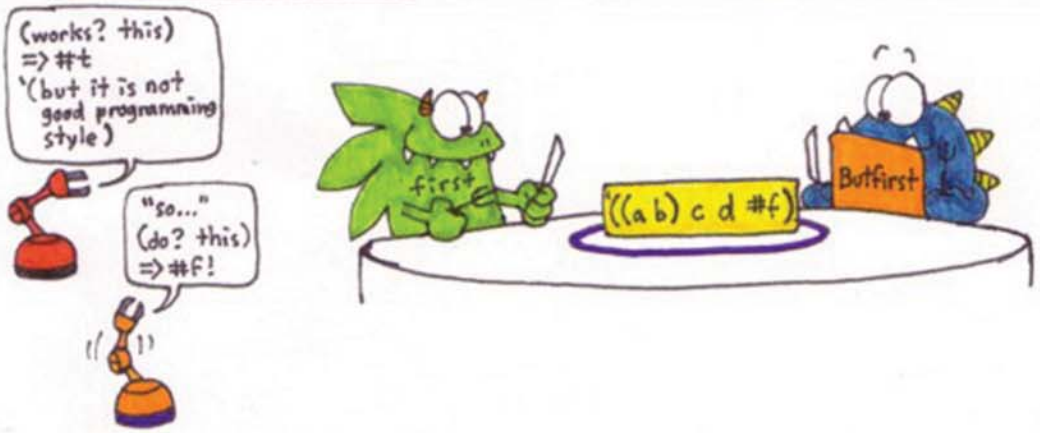
Giving Append A Non-List As An Argument:

Cons-ing A List To A Non-List (ie. Second Argument Isn't A List):

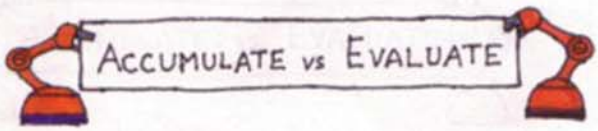
Using List Selectors On Non-Lists:



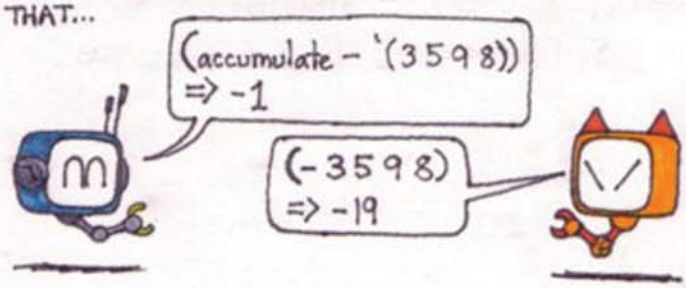
Using Word/Sentence Selectors on Lists (ie. Data Abstraction Faux Pas):



Ketrina Yim

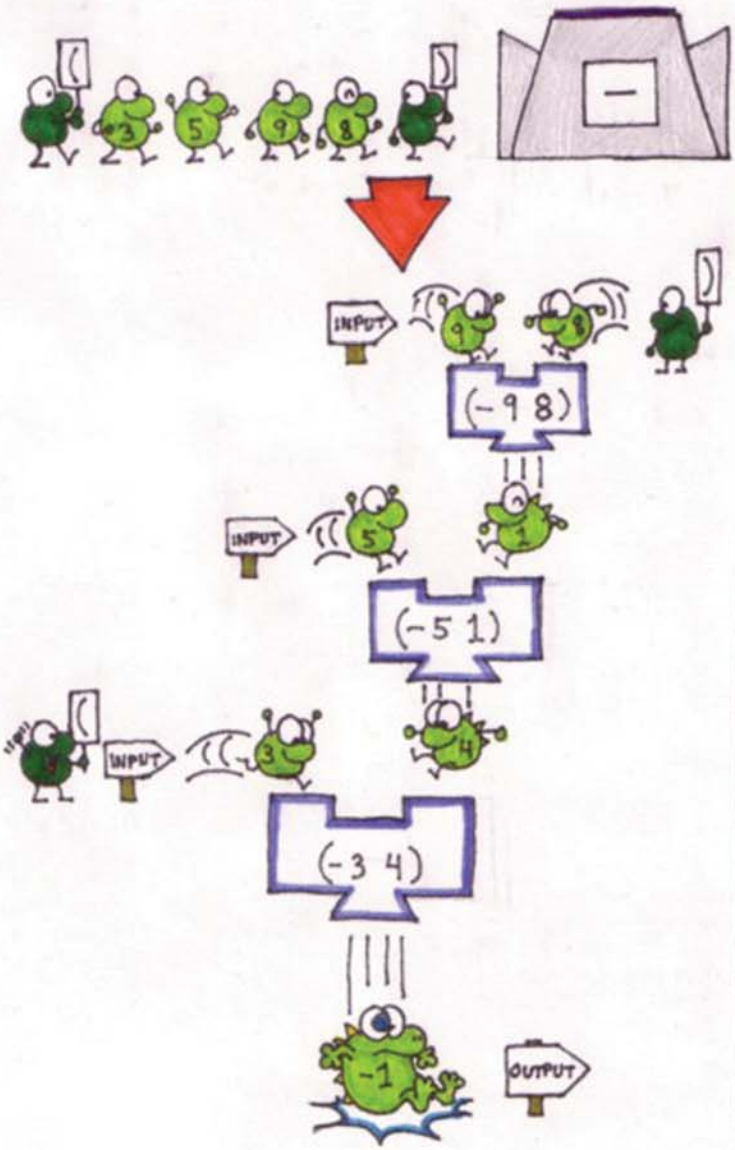


NOTICE THAT...

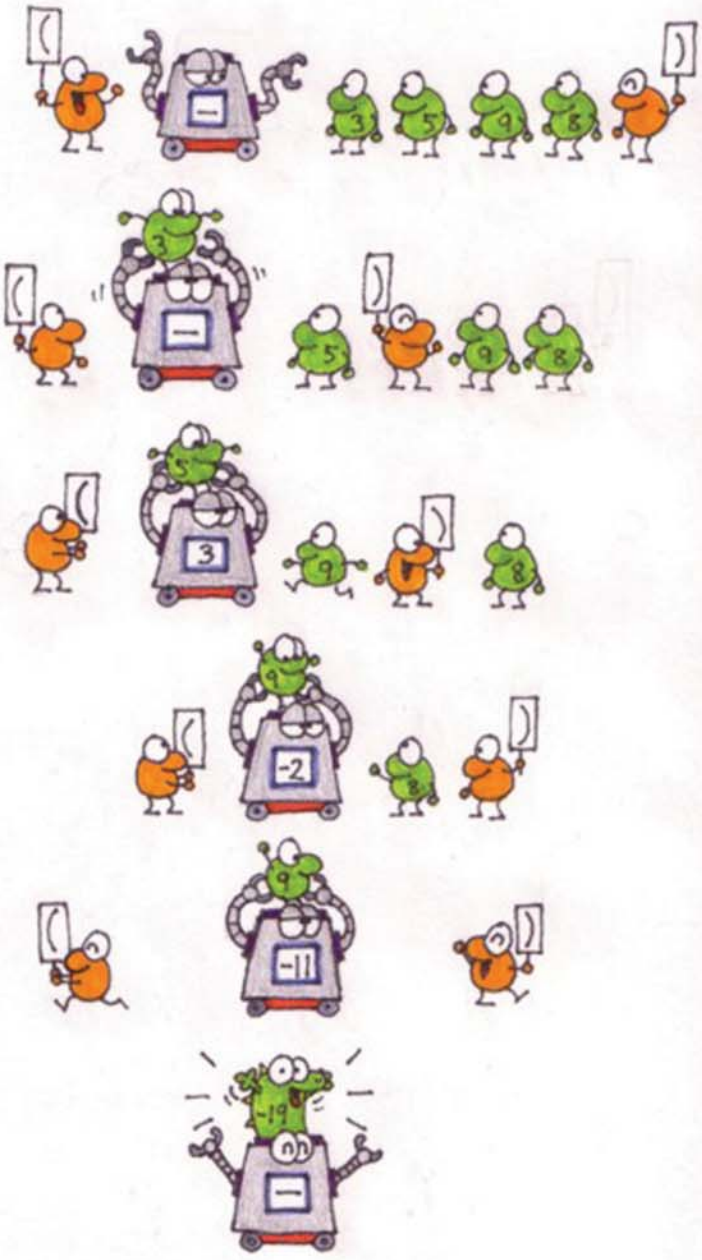


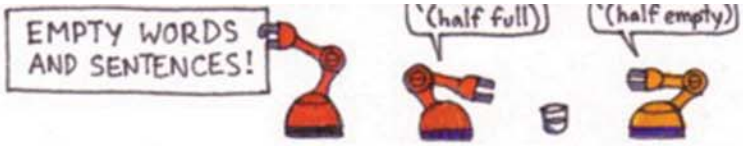
WHY ARE THE RETURN VALUES DIFFERENT?

BECAUSE ACCUMULATE PROCESSES DATA FROM RIGHT TO LEFT...



...WHILE THE ARITHMETIC EVALUATION GOES FROM LEFT TO RIGHT!

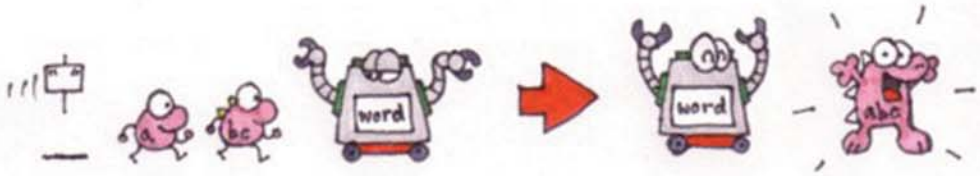




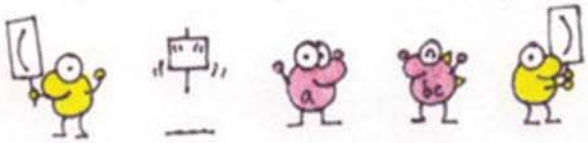
THIS IS AN EMPTY WORD.

TO MAKE THE EMPTY WORD "VISIBLE," IT MUST BE GIVEN DOUBLE QUOTES.

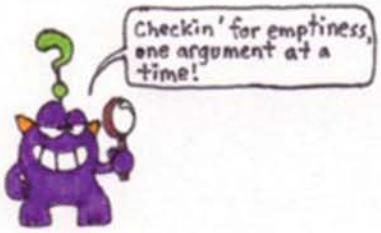
HOWEVER, WHEN YOU CALL WORD ON AN EMPTY WORD AND SOME NON-EMPTY WORDS, THE EMPTY WORD MUST DROP ITS QUOTES, SO IT "DISAPPEARS."



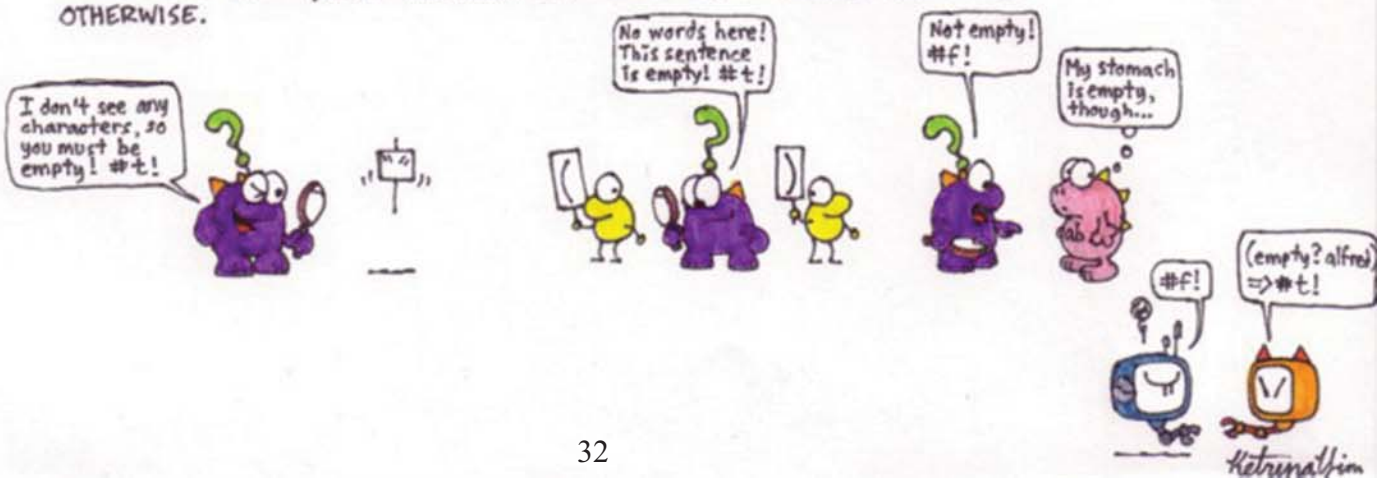
IT DOESN'T HAVE TO DROP THE QUOTES WHEN YOU CALL SENTENCE, THOUGH.

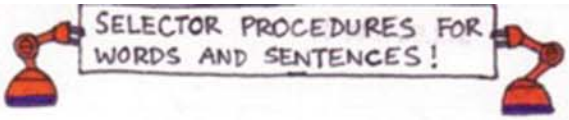


HOW DO YOU CHECK IF A WORD OR SENTENCE IS EMPTY? SIMPLY CALL FOR THE PREDICATE "EMPTY?"!

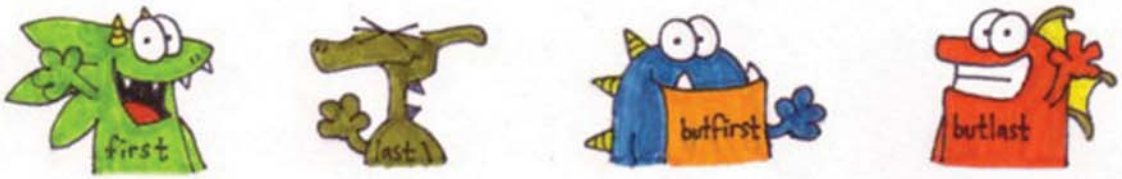


HOW DOES EMPTY? WORK? IT RETURNS TRUE IF THE INPUT WORD HAS NO CHARACTERS IN IT (OR IF THE INPUT IS A SENTENCE, NO WORDS) AND FALSE OTHERWISE.

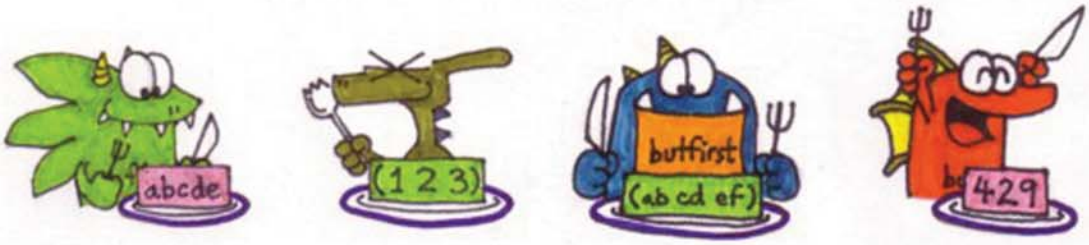




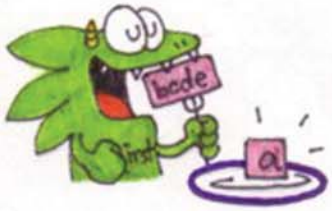
THESE ARE THE WORD/SENTENCE SELECTORS.



WHEN GIVEN A WORD OR SENTENCE...



FIRST RETURNS THE FIRST PART OF THE INPUT...

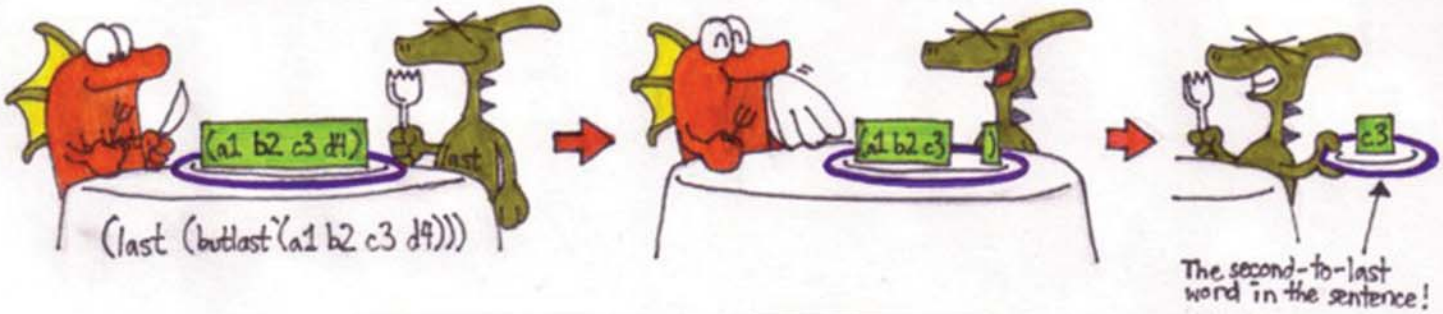


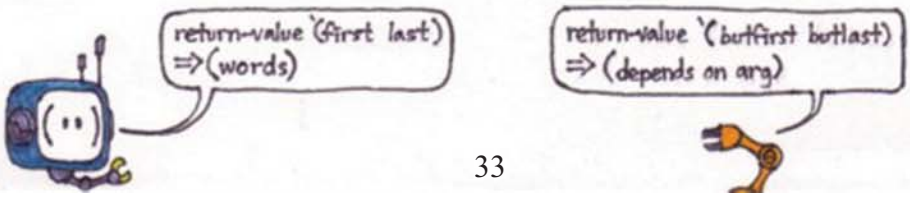
LAST RETURNS THE LAST PART...

BUTFIRST RETURNS ALL BUT THE FIRST PART...

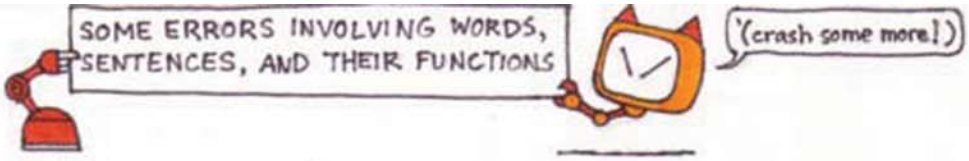
WHILE BUTLAST RETURNS ALL BUT THE LAST PART!

SELECT OTHER PARTS OF WORDS AND SENTENCES (LIKE THE SECOND OR THIRD ELEMENT) BY COMBINING SELECTORS!

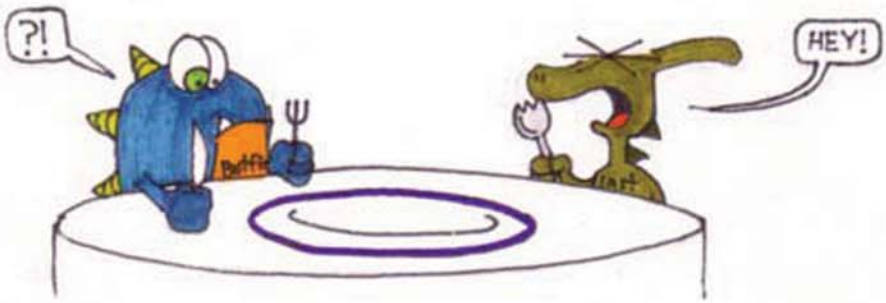




Katrina Thin



GIVING EMPTY WORDS OR EMPTY SENTENCES TO SELECTORS:



GIVING SENTENCES TO PROCEDURES THAT ONLY TAKE WORDS (OR VICE VERSA):

NOT QUOTING SENTENCES OR NON-NUMERICAL WORDS:

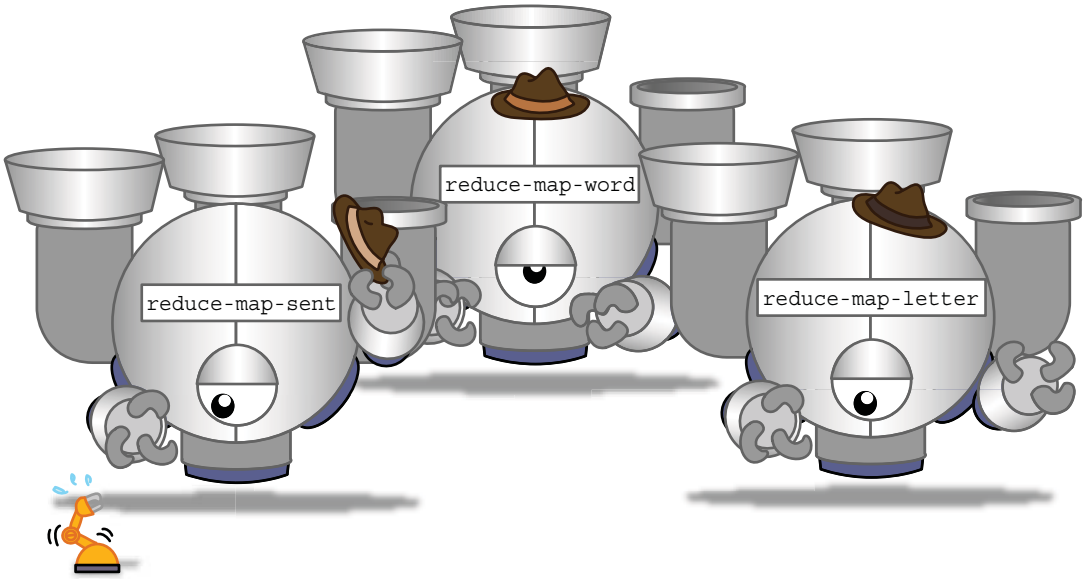
10.2 Appendix B: MapReduce for CS3

Course Description: CS3 introduces students to computer programming, emphasizing symbolic computation and the functional programming style. In this course, students write code in Scheme, a dialect of the LISP programming language.

The following illustrations were created to explain CS3's implementation of MapReduce.

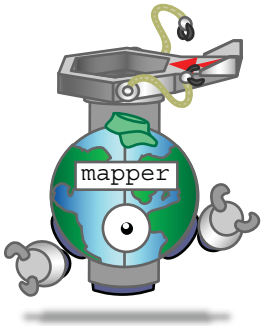
MapReduce

Parallelism and Functional Programming



MapReduce is a system that makes writing parallel code easier for programmers.

In CS3, we provide three functions that differ only in how they interpret data. Each takes three arguments:



The first of the arguments is a mapper, a function that takes a sentence, word or letter (depending on whether it was passed to `reduce-map-sent`, `reduce-map-word`, or `reduce-map-letter` respectively) and outputs some value. There are no constraints on what this value can be. (A mapper is the kind of function used as an argument to `map`.)

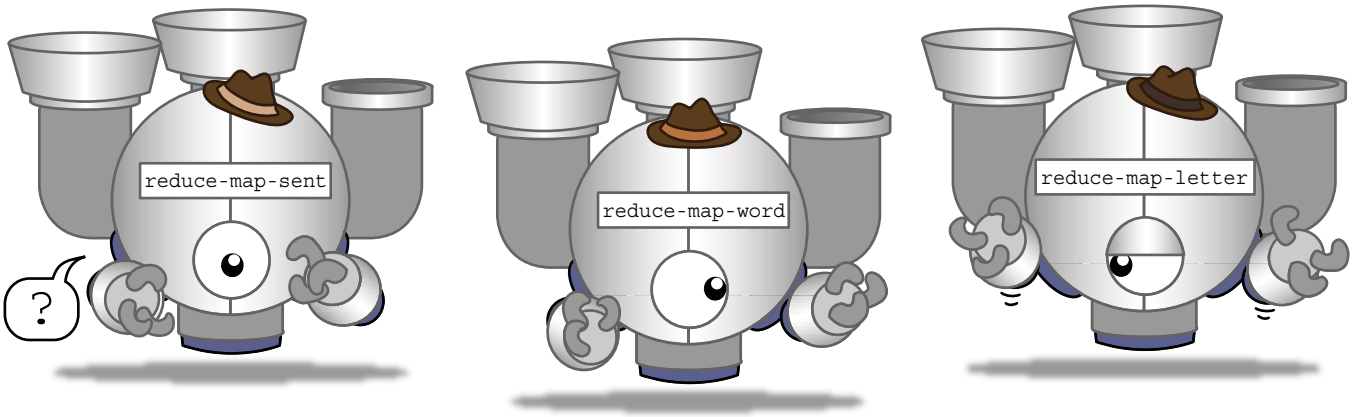
Next is the reducer, which collects and combines the values returned from the mappers, into one value.
(A reducer is the kind of function used as an argument to `reduce`.)

Finally, there's the vast body of data to be processed, specified by a filename, but encoded as a list. It's either a list of sentences, a list of words or a list of letters (depending on whether it was passed to `reduce-map-sent`, `reduce-map-word`, or `reduce-map-letter` respectively). If given a directory instead of a single file, MapReduce treats the input as a file composed of the concatenation of the individual files in the directory.

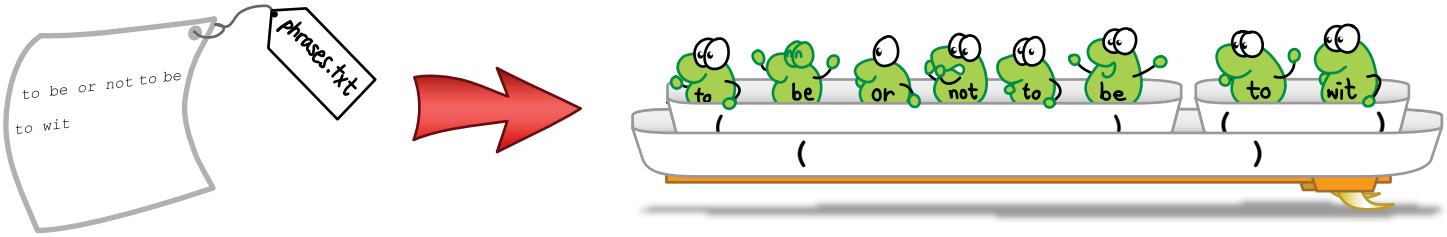
MapReduce

Why Have `sent`, `word`, and `letter`?

What exactly is the difference among these MapReduce functions?

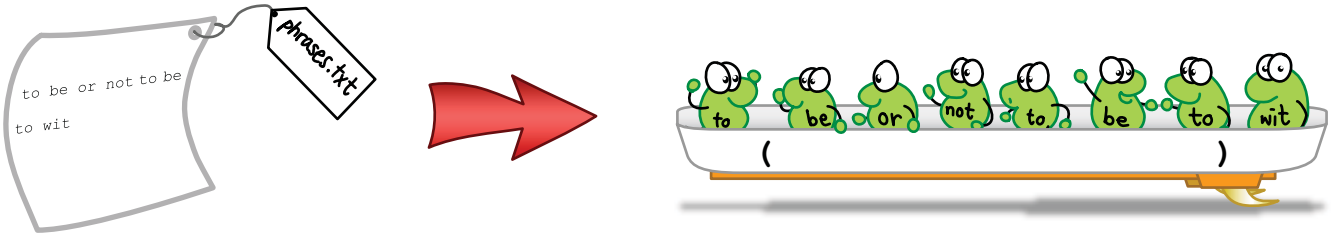


The answer lies in how they turn files into lists!



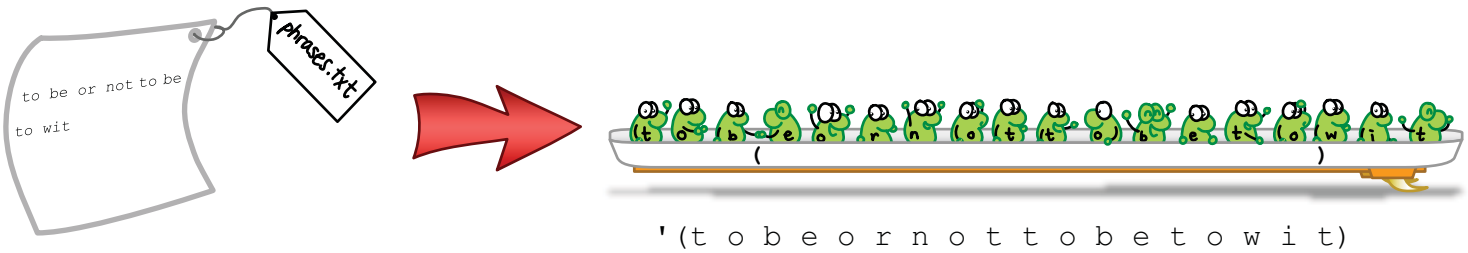
`'((to be or not to be) (to wit))`

`reduce-map-sent` creates a list of sentences, one sentence for each line of the file.



`'(to be or not to be to wit)`

`reduce-map-word` creates a list of the words in the file, ignoring carriage returns.

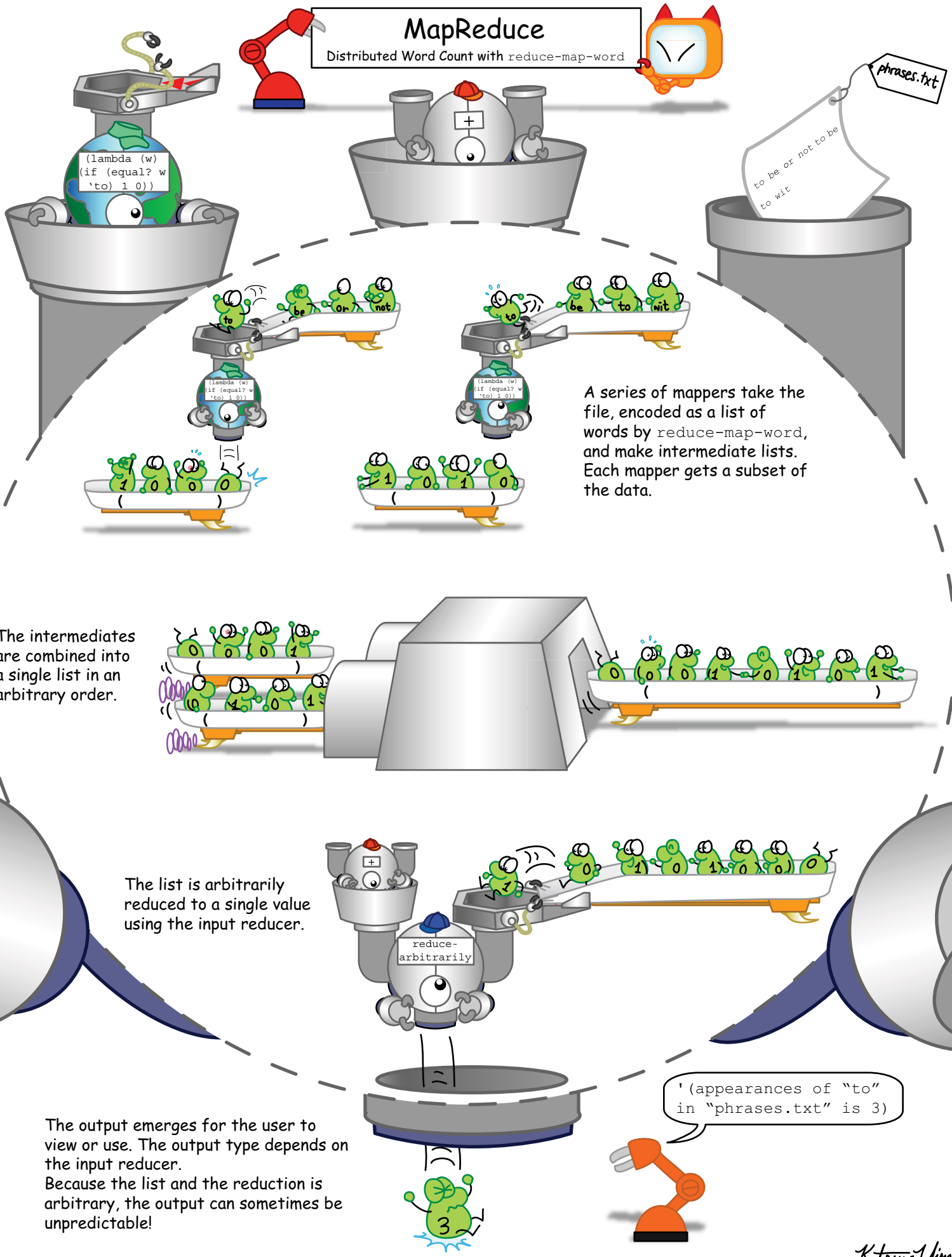


`'(t o b e o r n o t t o b e t o w i t)`

`reduce-map-letter` creates a list of the characters in the file, ignoring all whitespace.

MapReduce

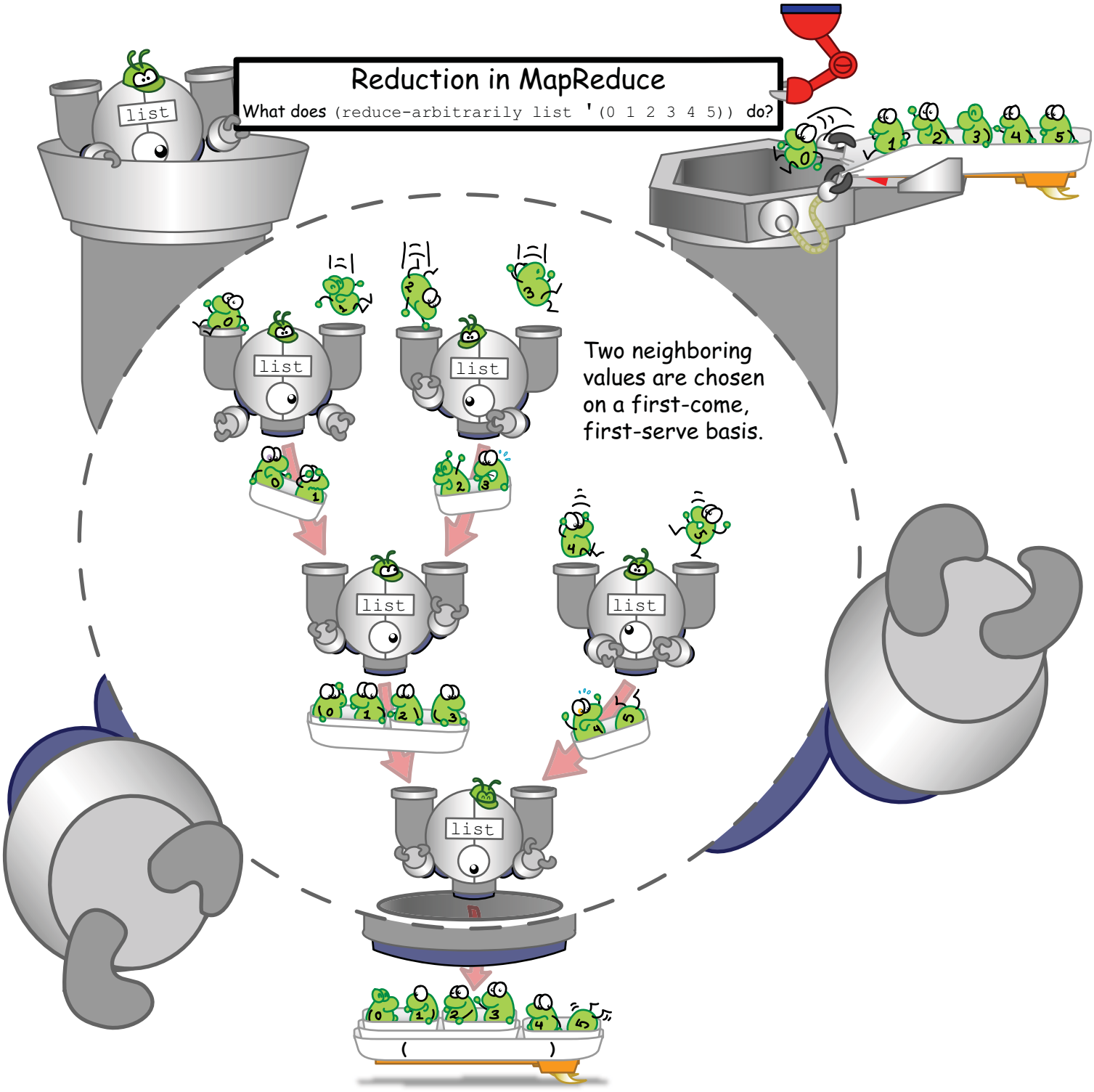
Distributed Word Count with `reduce-map-word`



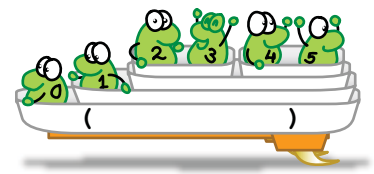
Katrina Yim

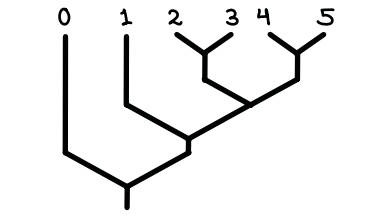
Reduction in MapReduce

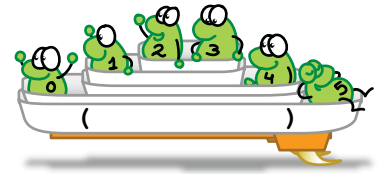
What does `(reduce-arbitrarily list '(0 1 2 3 4 5))` do?

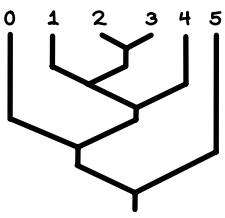


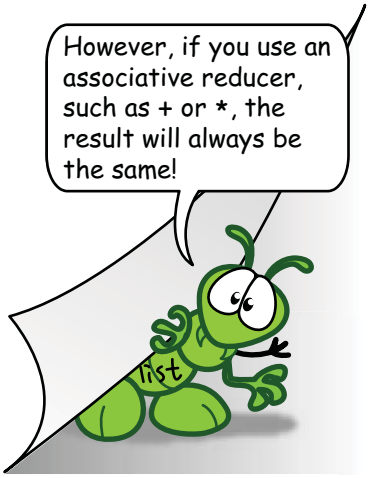
Depending on the order in which values are chosen, the same function call can give different results.









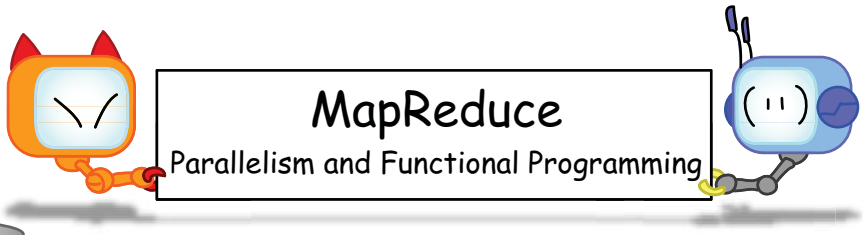


Ketrina Yim

10.3 Appendix C: MapReduce for CS61A

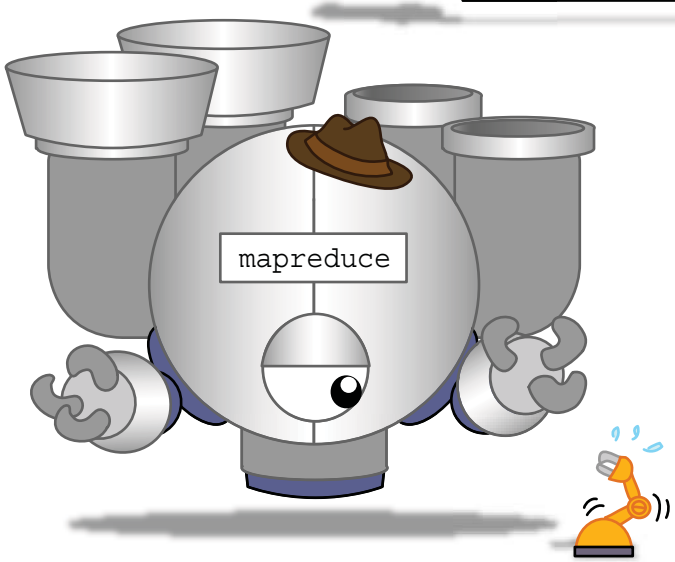
Course Description: CS61A is an introduction to programming and computer science, exposing students to techniques of abstraction at several levels: within a programming language, using higher-order functions, manifest types, data-directed programming, and message-passing; and between programming languages, using functional and rule-based languages as examples. Programming projects and assignments are done in Scheme.

The following illustrations were created to explain CS61A's implementation of MapReduce:



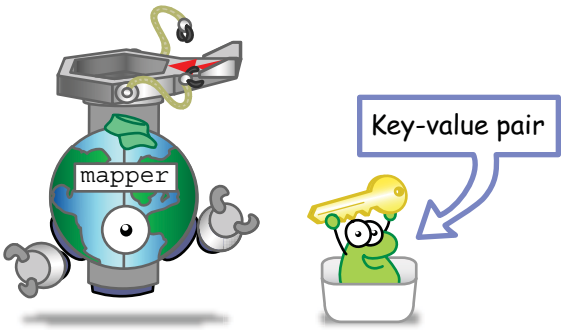
MapReduce

Parallelism and Functional Programming



MapReduce is a system that makes writing parallel code easier for programmers.

It takes four arguments:



The first argument is the mapper, a function that takes one key-value pair and outputs a list of intermediate key-value pairs.
(A mapper is the kind of function used as an argument to `map`.)

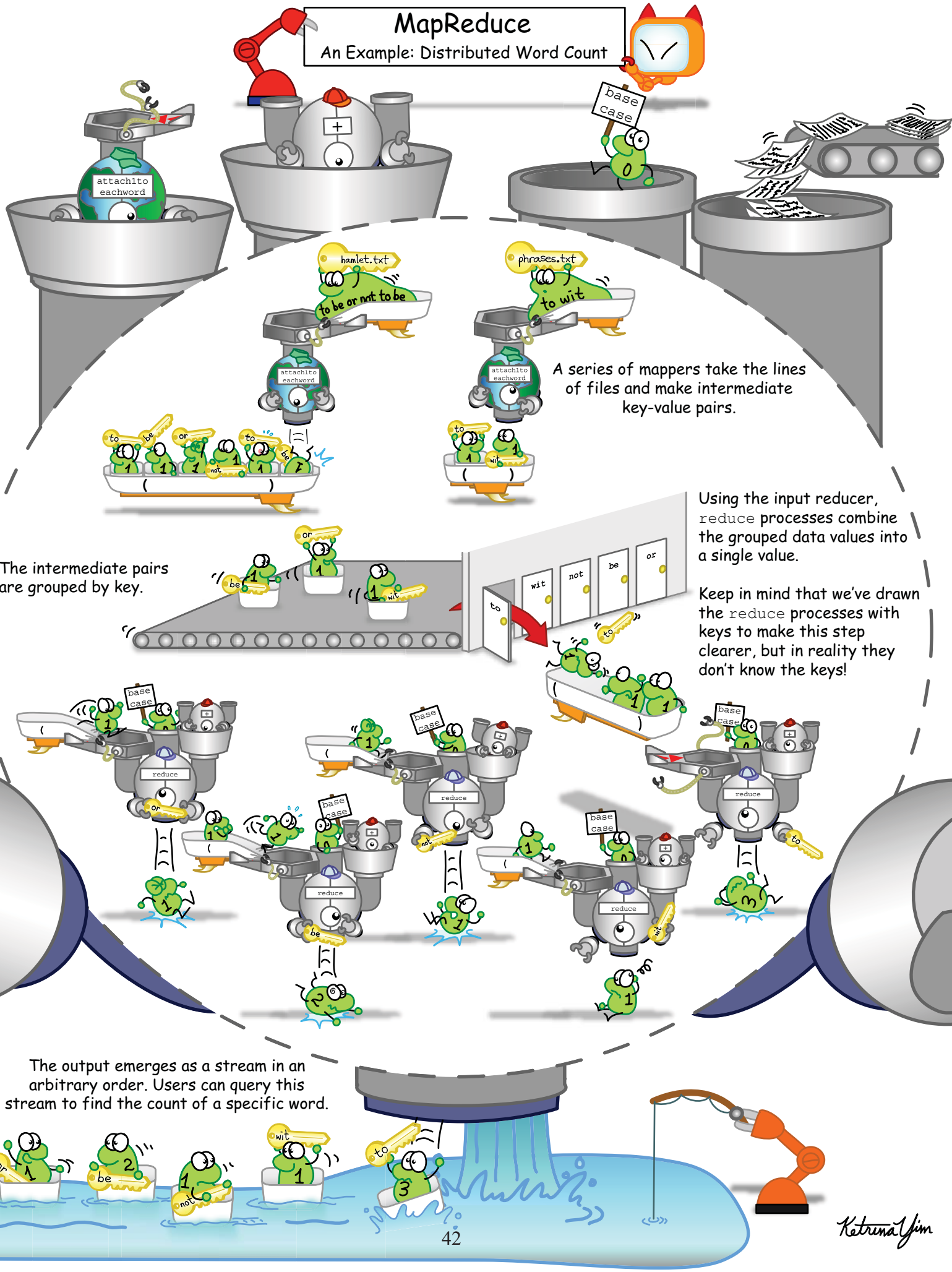
Next is the reducer, which collects and combines values. It plays a vital role in turning intermediate key-value pairs into output.
(A reducer is the kind of function used as an argument to `accumulate`, also known as `reduce`)

Then comes the base case, which is usually the "identity" of the reducer function given to `accumulate`.

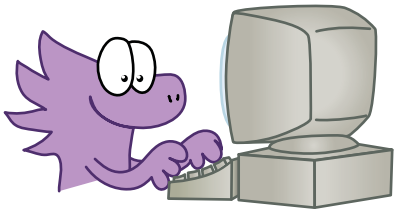
Finally, there's the vast body of data to be processed, encoded as a stream of key-value pairs.

MapReduce

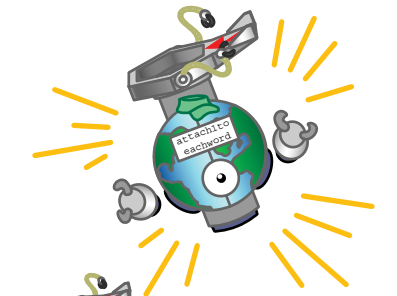
An Example: Distributed Word Count

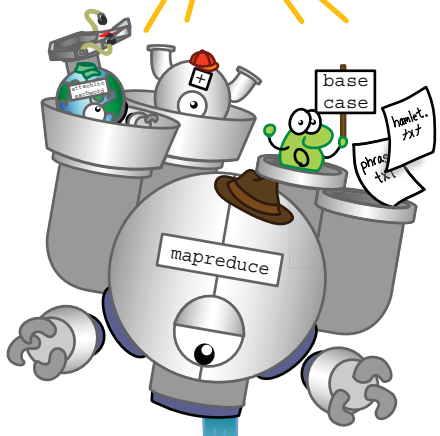


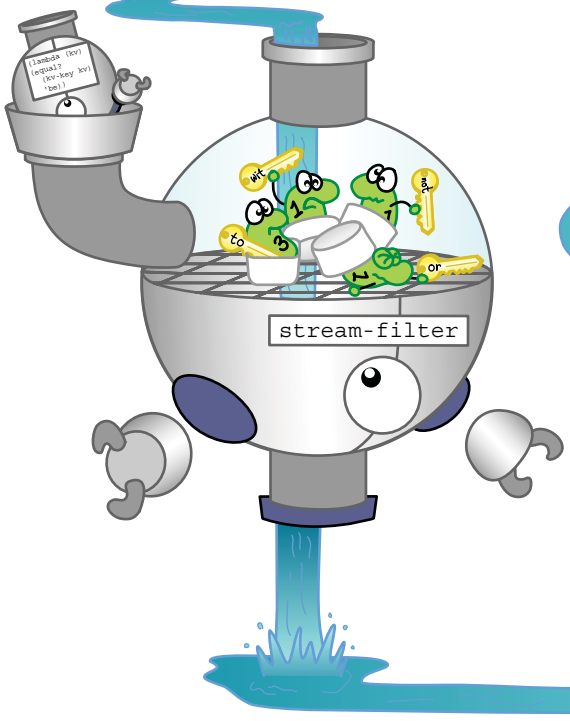
MapReduce
The Wordcount in Code

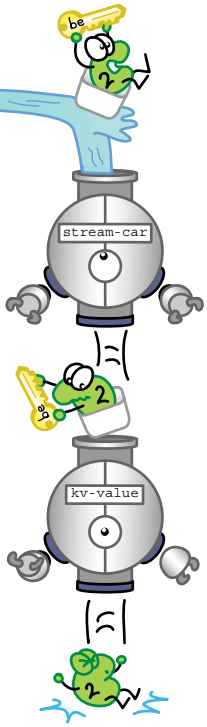


```
unix% ls /docs
hamlet.txt phrases.txt

unix% stk
;; Define the mapper
;; Typical kv-pair:
;; (phrases.txt . (to wit))
;; Output:
;; ((to . 1) (wit . 1))
> (define (attach-1-to-each-word document-line-kv-pair)
    (map (lambda (wd-in-line)
           (make-kv-pair wd-in-line 1))
         (kv-value document-line-kv-pair)))
attach1toeachword

;; Invoke mapreduce as a distributed word count
> (define wordcounts
    (mapreduce attach-1-to-each-word + 0 "/docs"))
wordcounts

;; Display the elements of the output stream
> (show-stream wordcounts 5)
((or . 1) (be . 2) (not . 1) (wit . 1) (to . 3))

;; Query the stream for the count of the word "be"
> (kv-value (stream-car (stream-filter
    (lambda (kv) (equal? (kv-key kv) 'be)) wordcounts)))
2
```









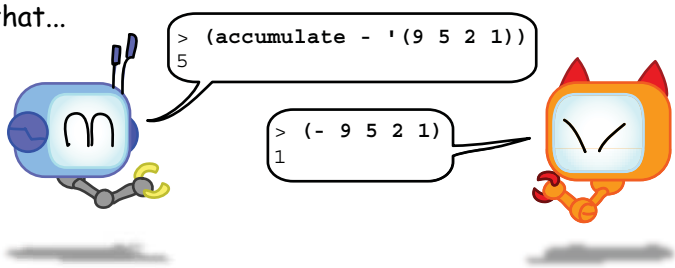
10.4 Appendix D: Other Illustrations for CS3 and CS61A

The following illustrations are applicable to both CS3 and CS61A, unless stated otherwise, because they use Scheme and there is overlap in the topics covered. Many of these are revised versions of the precursors seen in Appendix A.

Page 45: Explaining the differences between accumulate and standard evaluation.
Page 46: Explaining empty words and sentences and introducing the `empty?` predicate.
Page 47: Introducing the word and sentence selectors.
Page 48: Common mistakes students make when using the word and sentence datatypes.
Page 49: Explaining how functions can be the input and output of other functions.
Page 50: Introducing higher-order functions.
Page 51: Introducing the list constructors: `cons`, `append`, and `list`.
Page 52: Common mistakes students make when using lists (specific to CS3).
Page 53: Common mistakes students make when using lists (specific to CS61A).

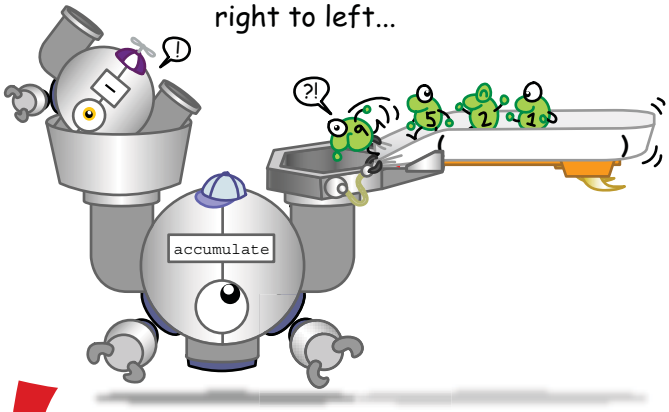
Arithmetic Operations: `accumulate` vs Evaluate

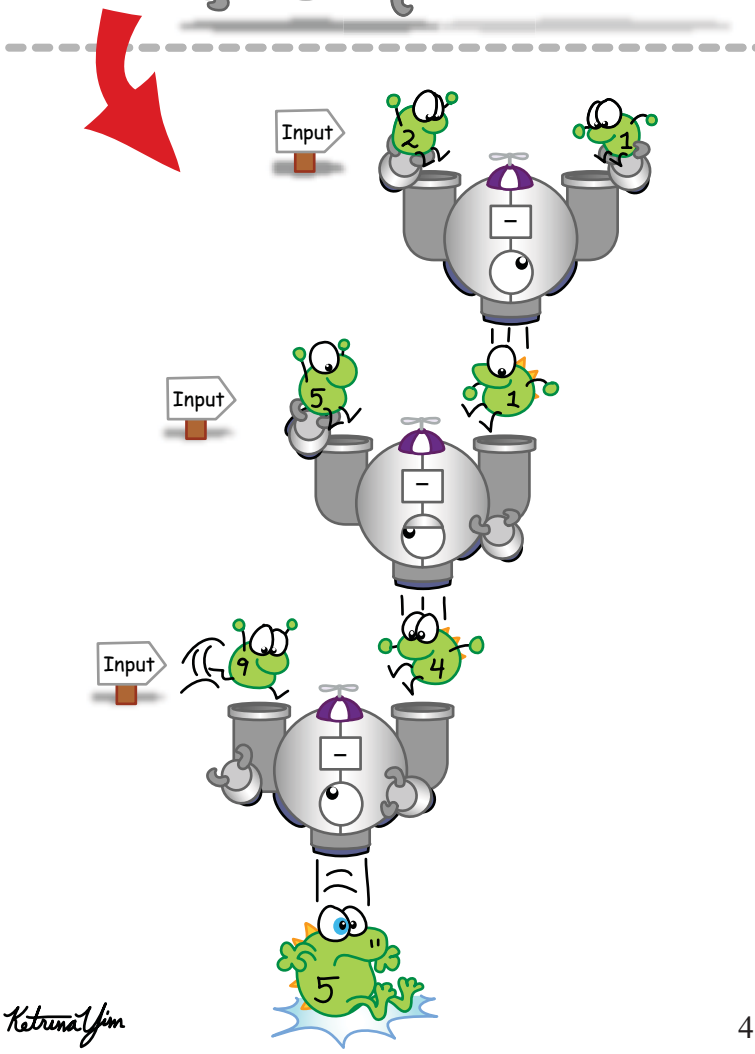
Notice that...



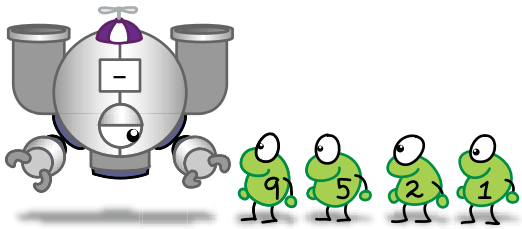
Why are the return values different?

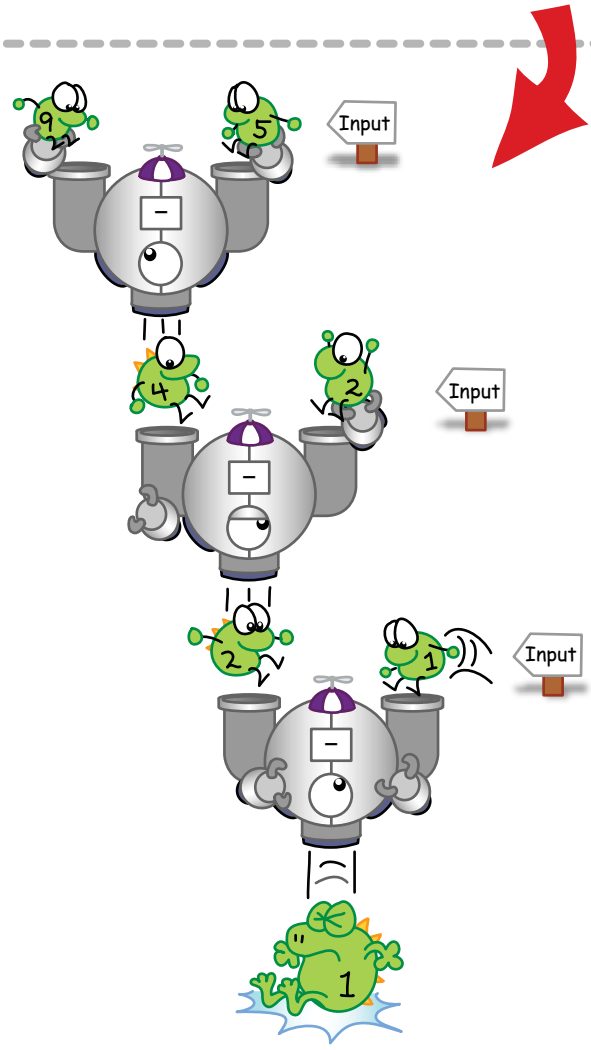
Because `accumulate` processes data from right to left...



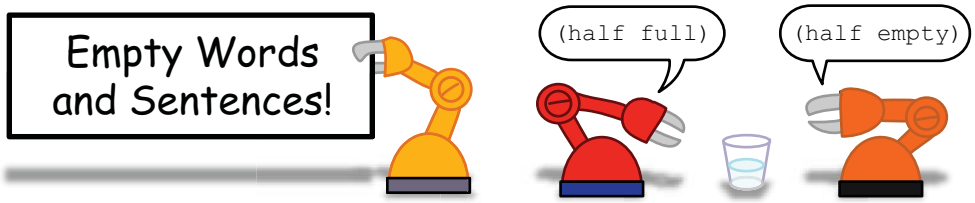


...while the arithmetic evaluation goes left to right!





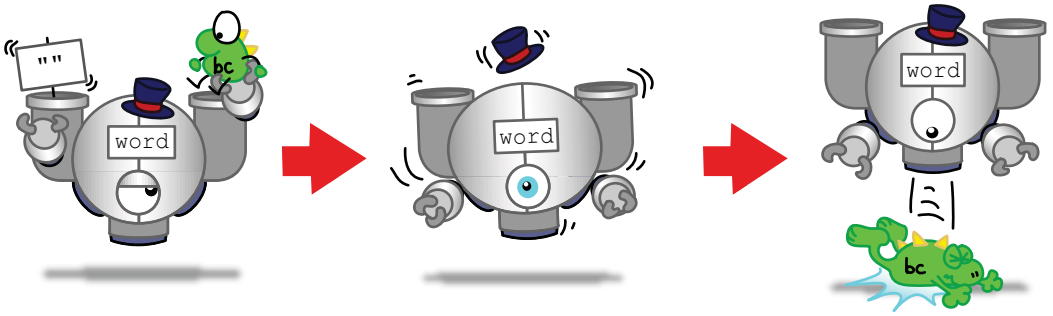
Katrina Yim



This is an empty word.

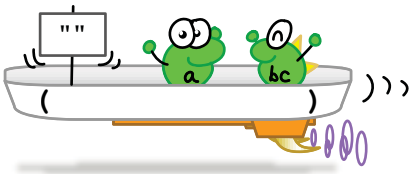
It must be given double quotes, to make it "visible".

Empty words are the identity of the `word` function, much like zero is the identity of +. Thus, empty words seem to disappear when you call `word` on them with non-empty words.



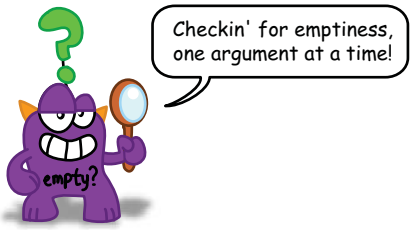

```
> (word "" 'bc)
bc
```

You'll still see the empty words when you use `sentence`, though.

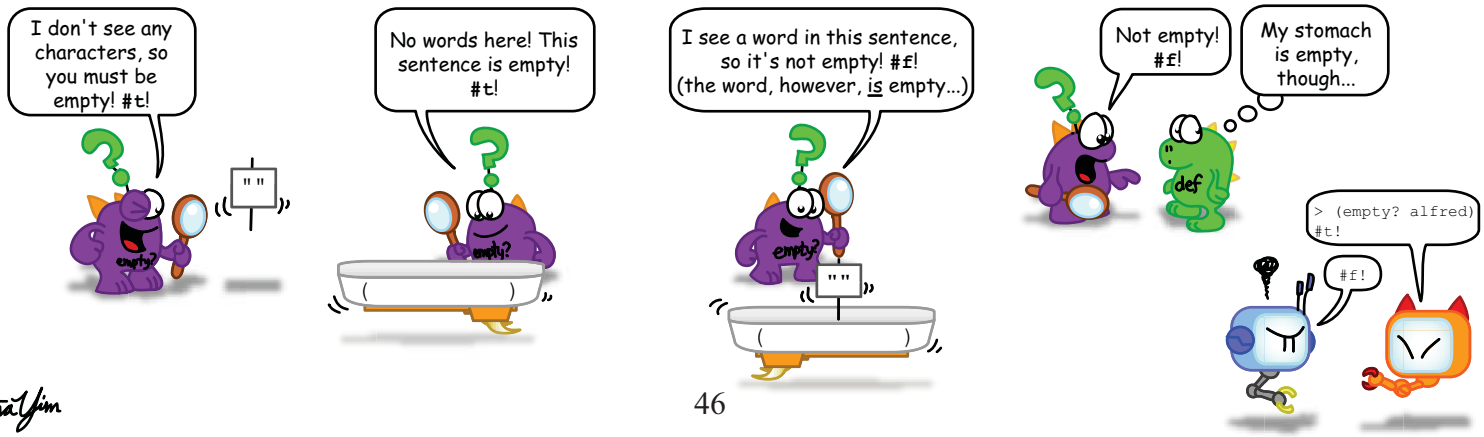



```
> (sentence "" 'a 'bc)
("" a bc)
```

How can you tell when a word or sentence is empty? Simply call the predicate `empty?`!



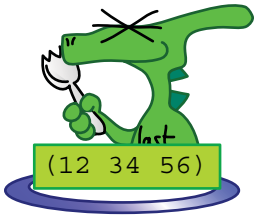
How does `empty?` work? It returns true if the input is a word with no characters or a sentence with no words and false otherwise.

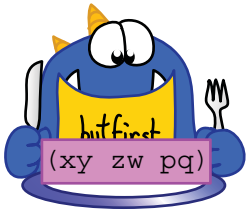


Selector Procedures for Words and Sentences!

These are the selectors for words and sentences.

When given a word or a sentence...



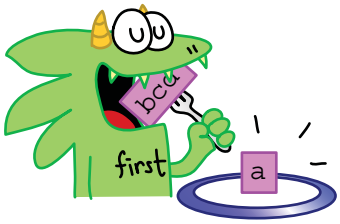


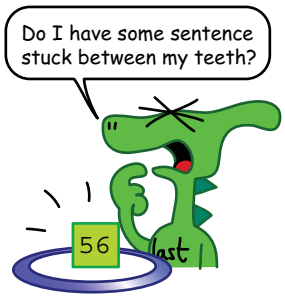
`first` returns the first part of the input...

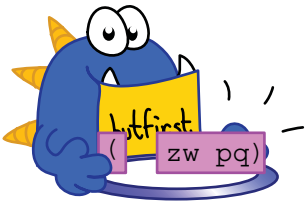
`last` returns the last part...

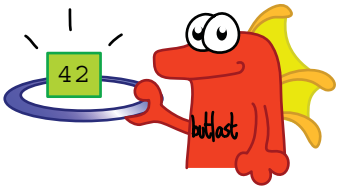
`butfirst` returns all but the first part of the input...

And `butlast` returns all but the last part!

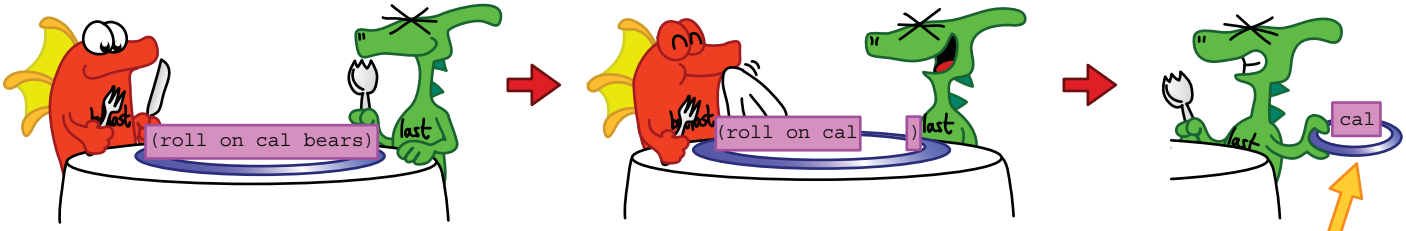






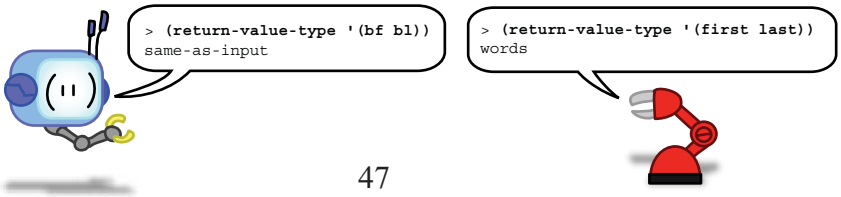


Select other parts of words and sentences, such as the second or third element, by combining selectors!



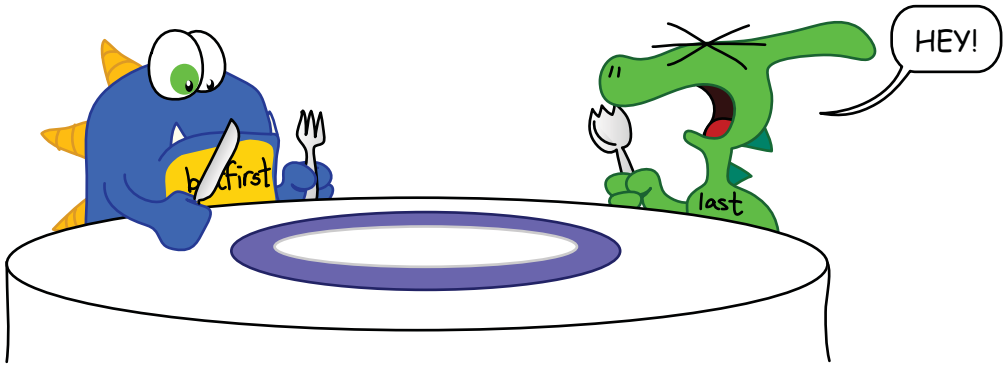

```
> (last (butlast '(roll on cal bears)))
cal
```

One thing to note here is that the selectors don't actually change the input they're given. They make a copy of the input and do their work on the copy.

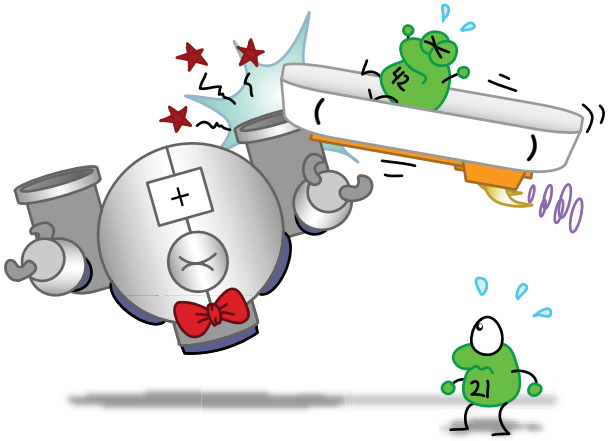


Common Errors Involving Words, Sentences, and Their Functions

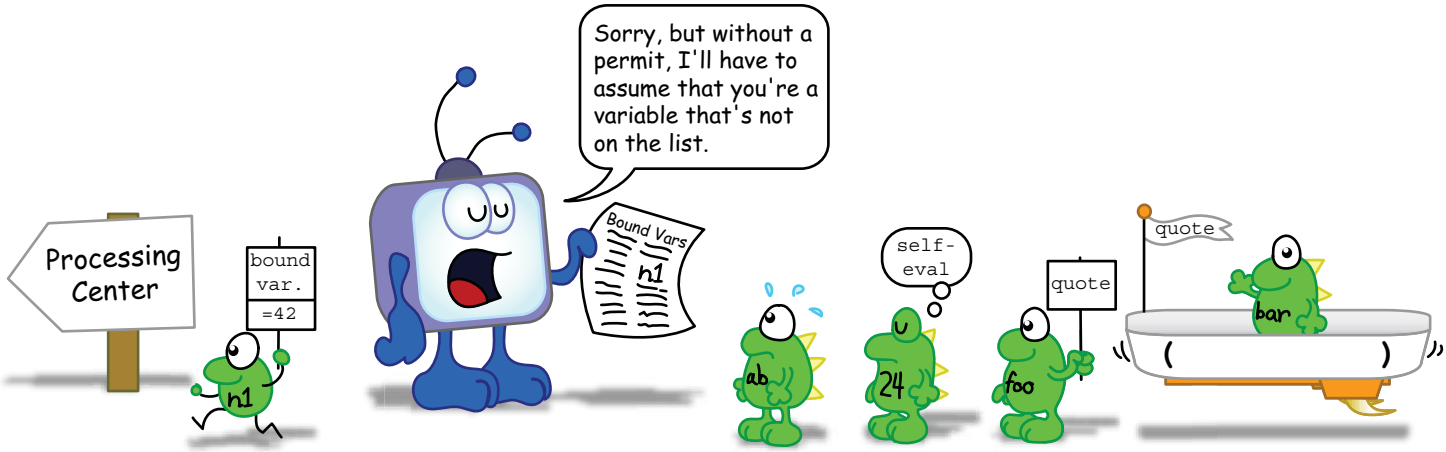
Giving empty words or sentences to selectors:

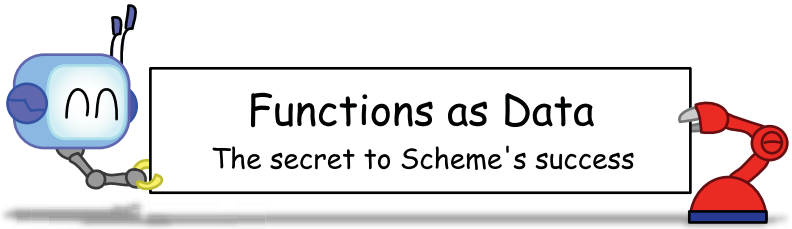


Giving sentences to procedures that only take words (and vice versa):



Not quoting sentences or non-numerical words:

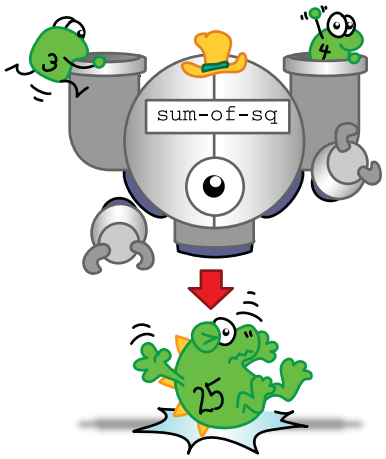




Functions as Data

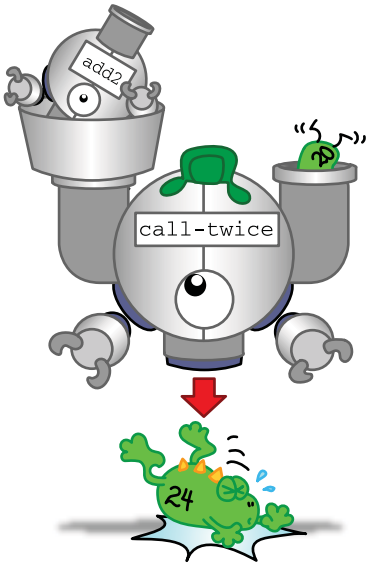
The secret to Scheme's success

Most functions take in data as arguments and output data.



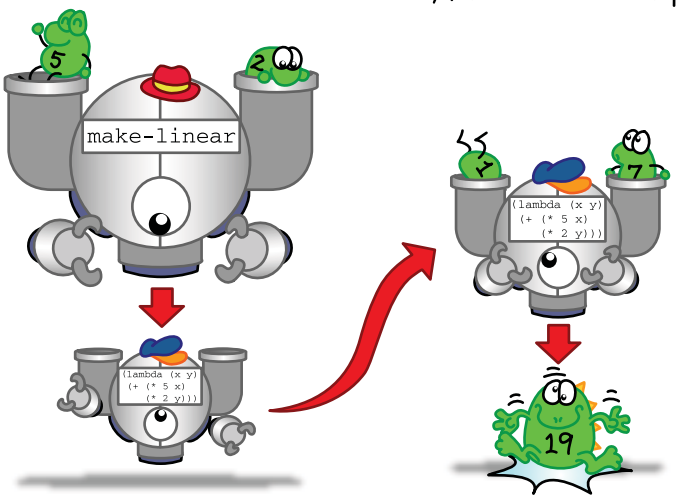

```
;; Compute sum of squares
> (define (sum-of-sq x y)
      (+ (* x x) (* y y)))
sum-of-sq

> (sum-of-sq 3 4)
25
```

But Scheme is great because it's easy to pass functions as arguments, like they're data!




```
> (define (add2 n) (+ n 2))
add2

;; Invoke a function twice
> (define (call-twice func x)
      (func (func x)))
call-twice

> (call-twice add2 20)
24
```

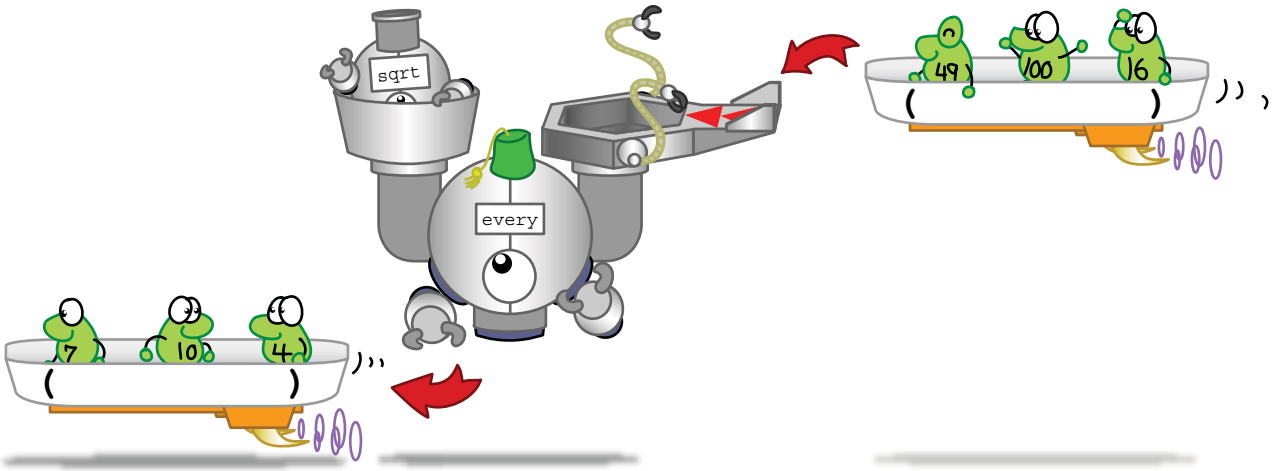
Also, functions can output new functions!



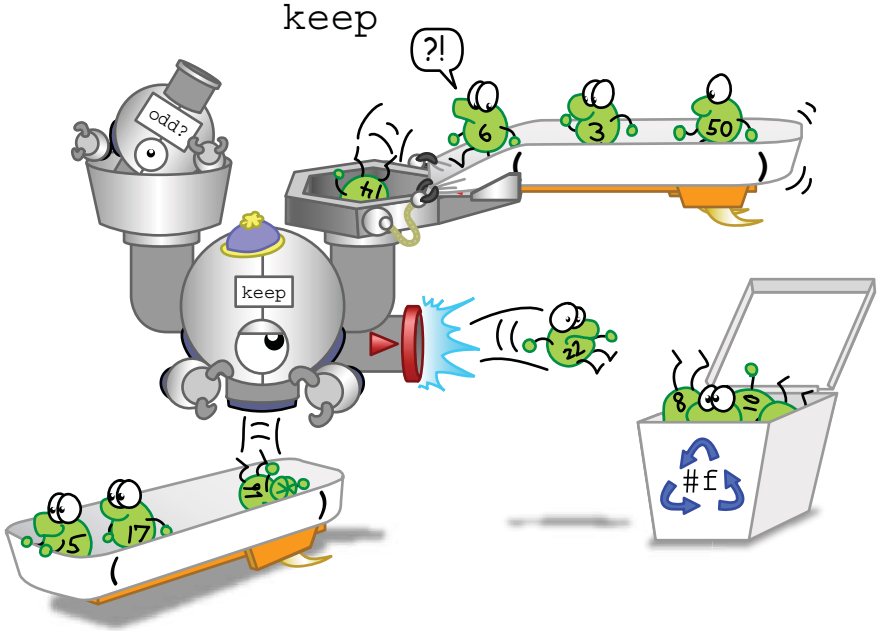

```
;; Generate a linear equation function
> (define (make-linear a b)
      (lambda (x y) (+ (*a x) (* b y))))
make-linear

> ((make-linear 5 2) 1 7)
19
```

Higher-Order Procedures for Sentences!

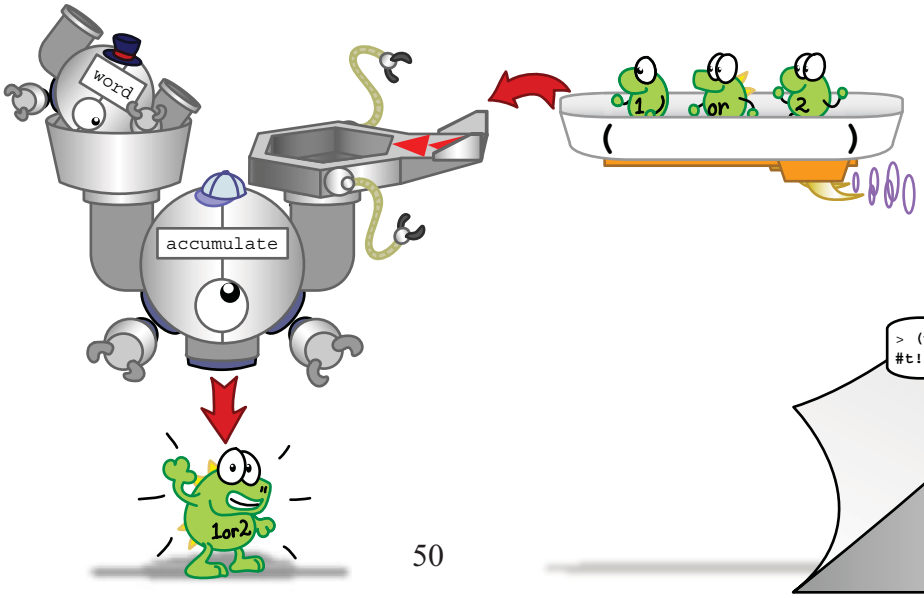
every

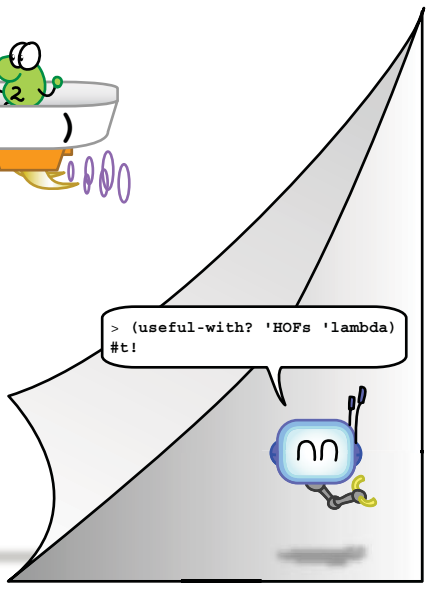


keep



accumulate

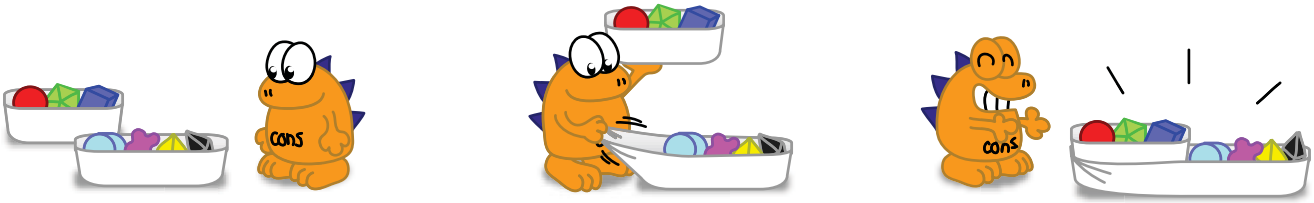




List Constructors

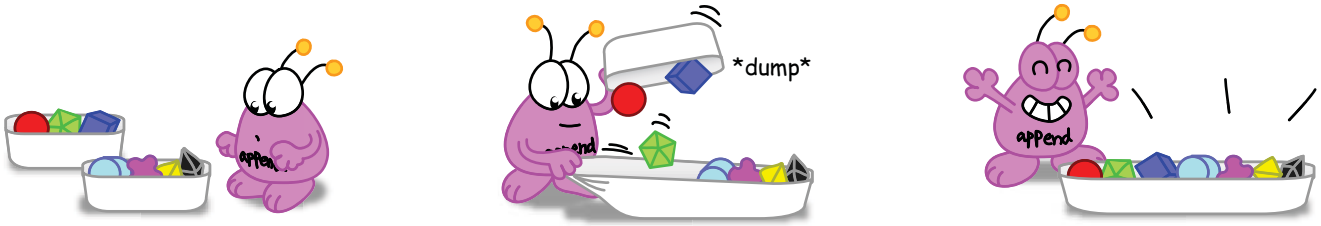
When given two lists...

cons



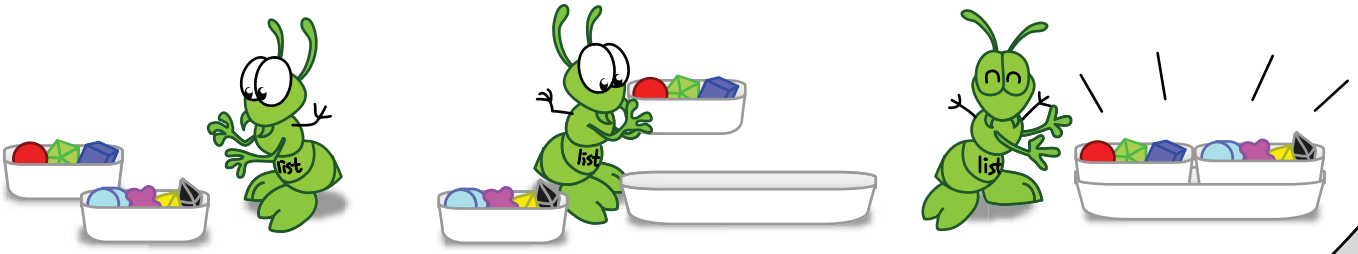

```
> (cons '(r g b) '(c m y k))
((r g b) c m y k)
```

append



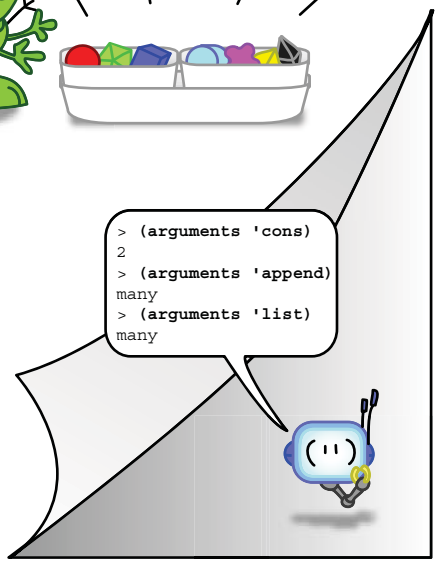

```
> (append '(r g b) '(c m y k))
(r g b c m y k)
```

list

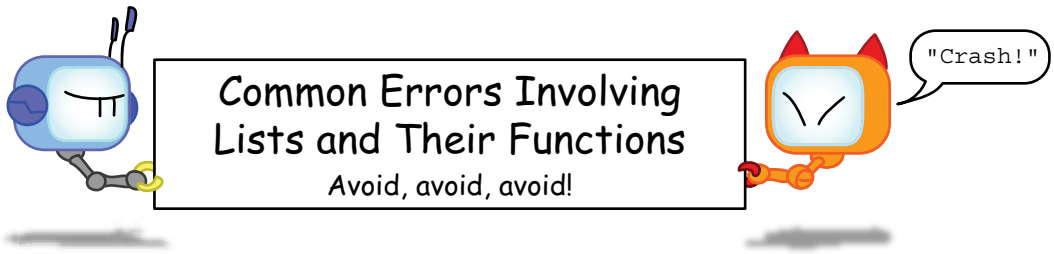



```
> (list '(r g b) '(c m y k))
((r g b) (c m y k))
```


Keep in mind that the list constructors don't change the input given to them. They make copies of the input and work with the copies.

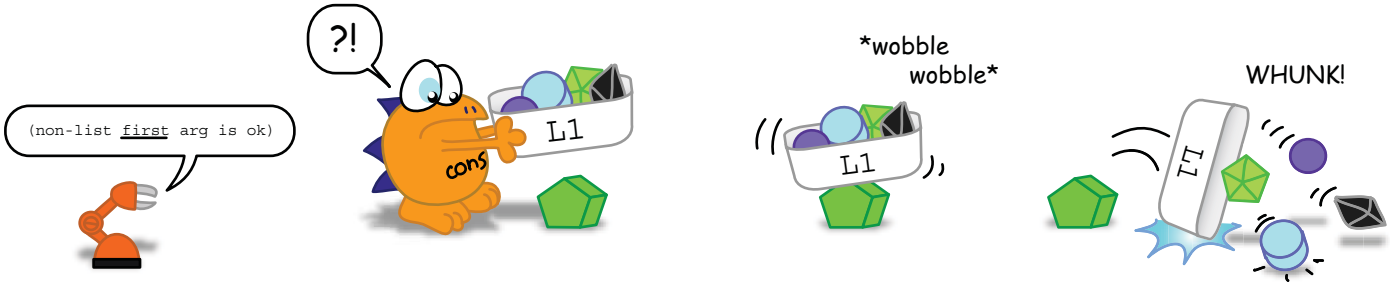


Katrina Yim

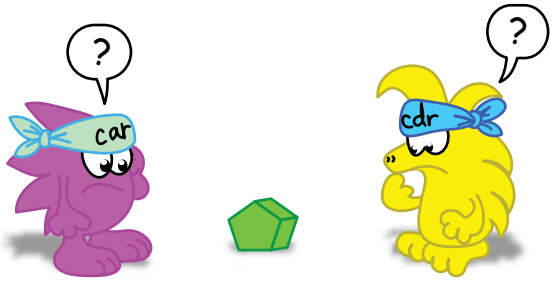


Giving `append` a non-list as an argument:

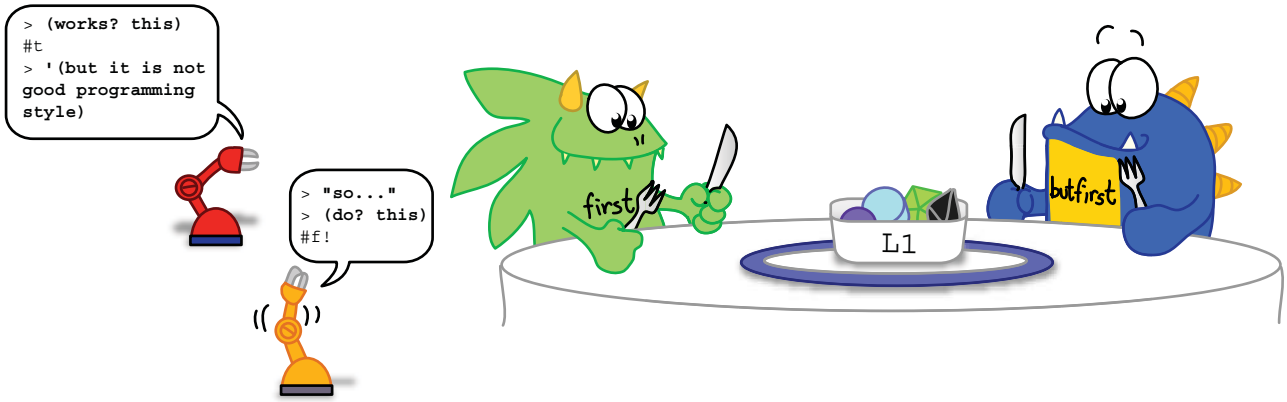
`cons`-ing a list to a non-list (i.e. second argument isn't a list):



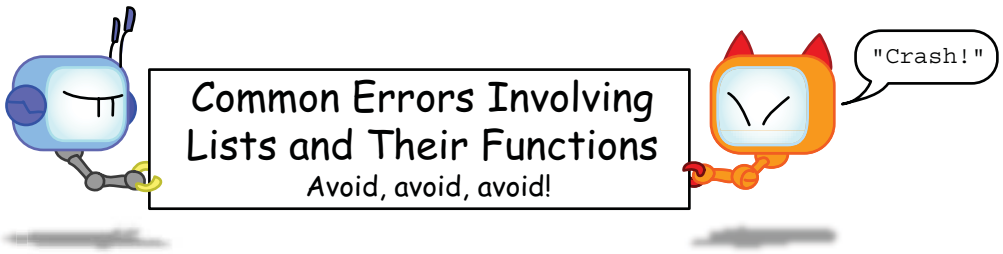
Using list selectors on non-lists:



Using word and sentence selectors on lists (i.e. data abstraction faux pas):



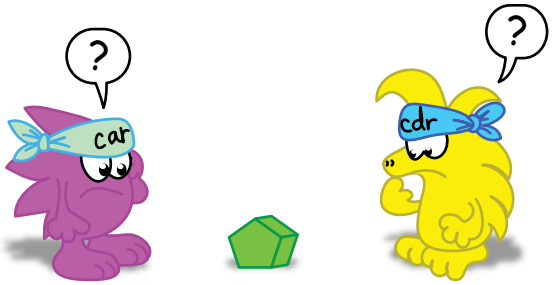
Ketrina Yim



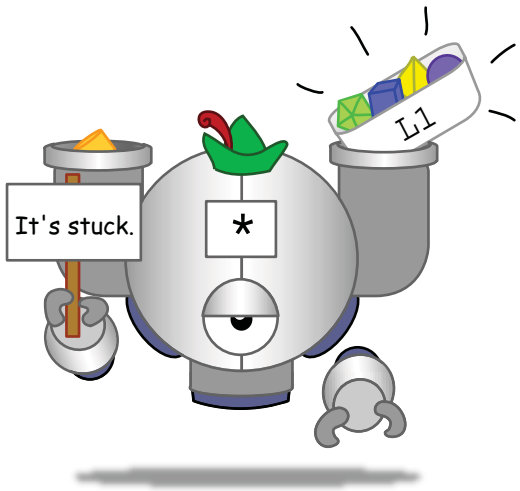
Giving `append` a non-list as an argument:

`cons`-ing a list to a non-list (i.e. second argument isn't a list):

Using list selectors on non-lists:



Giving lists to a function that takes only non-lists as arguments (or vice versa):



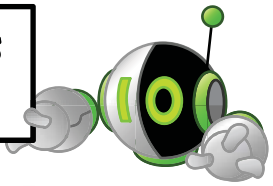
Katrina Yim

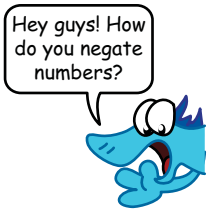
10.5 Appendix E: Illustrations for CS61C

Course Description: CS61C covers the internal organization and operation of digital computers. Topics covered include machine architecture support for high level languages and operating systems, elements of computer logic and CPU design, pipelined architecture and other aspects of machine parallelism, and the tradeoffs involved in fundamental architectural design decisions. Programming assignments are primarily in the C programming language.

The following illustrations cover several major topics of CS61C, including binary integers, floating-point numbers, pointers and arrays, and caching.

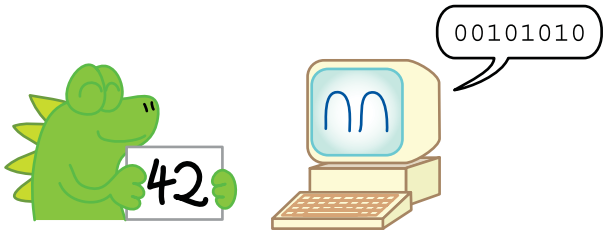
Binary Representations for Integers





In the early days of computing, designers made computers express numbers using **unsigned binary**.

And they were content...

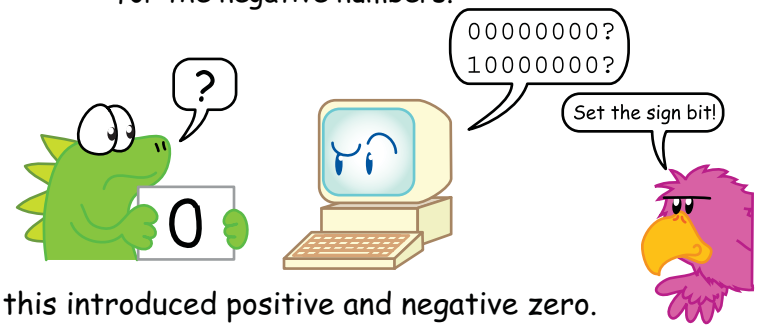


Until there were negative numbers.

To include negative numbers, designers came up with **sign magnitude**.

That took care of the negative numbers...

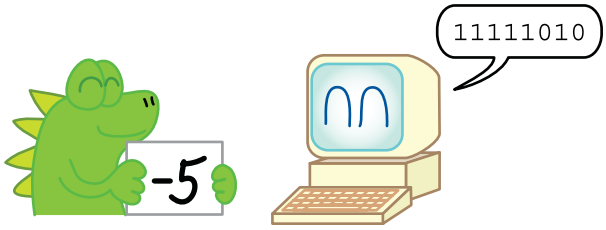
But the computer had to count backwards for the negative numbers.



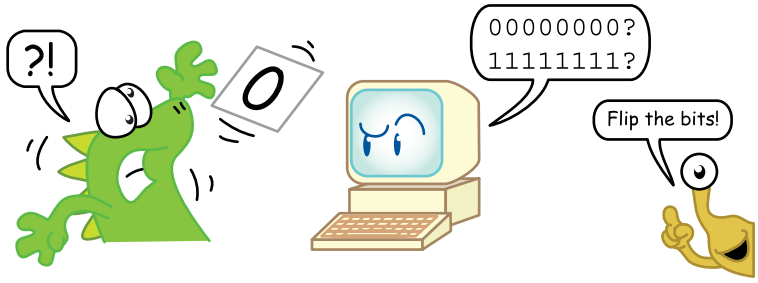
Plus, this introduced positive and negative zero.

Then designers created **one's complement**.

Now computers only had to count in one direction...



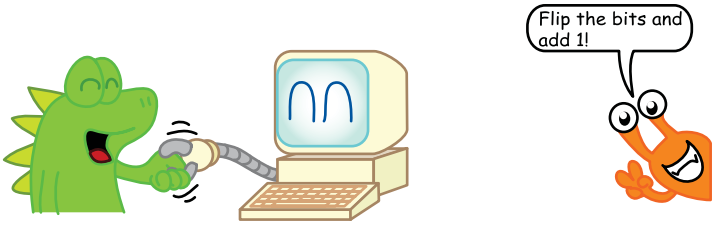
But there were still two zeroes!



Finally, designers developed **two's complement**.

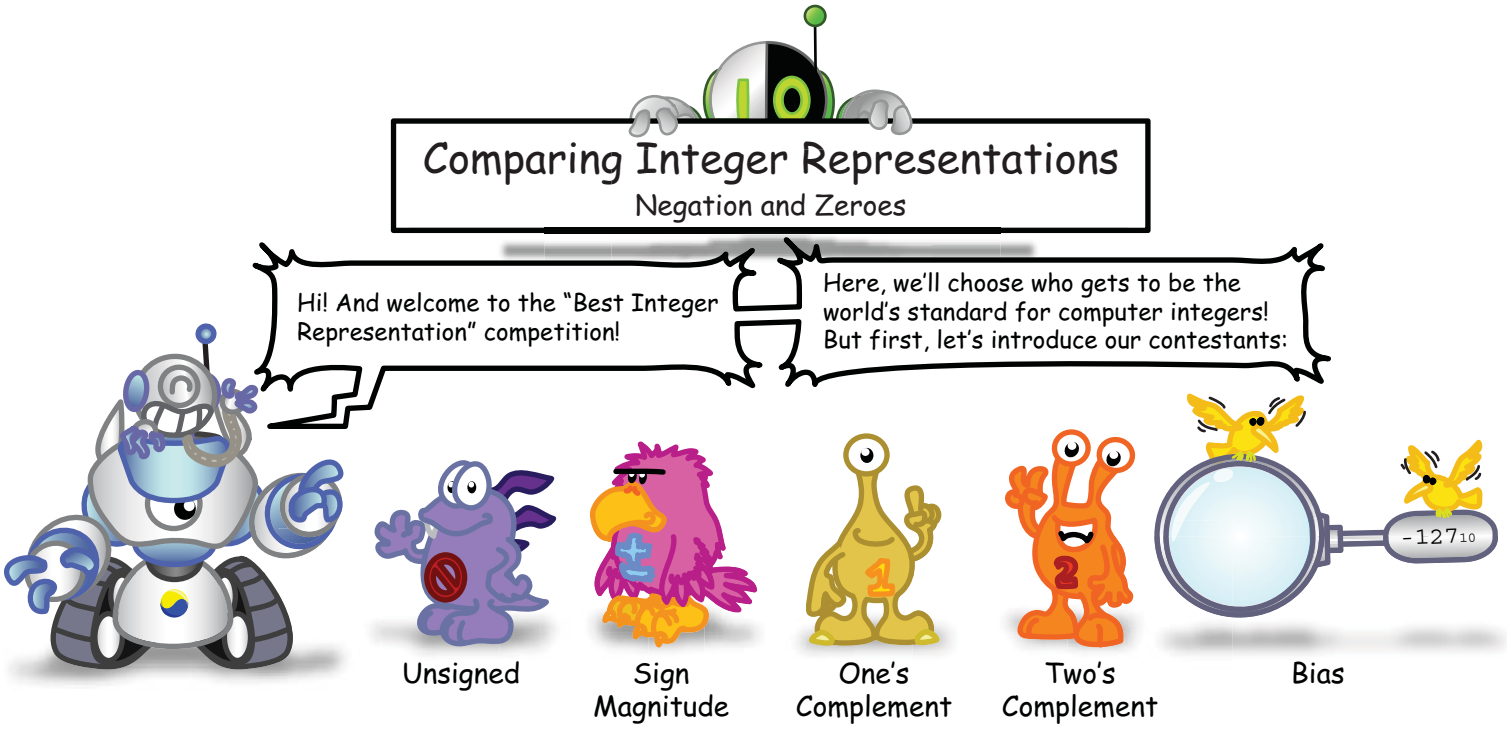
Now, there was only one zero...

And they were content.



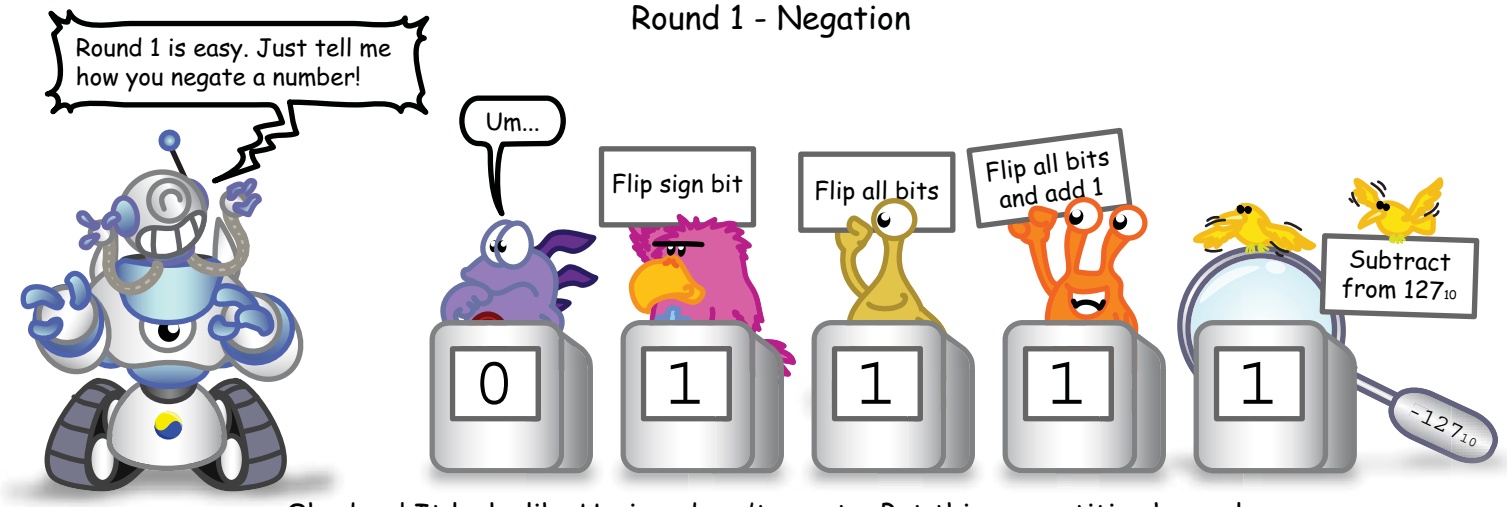
Comparing Integer Representations

Negation and Zeroes



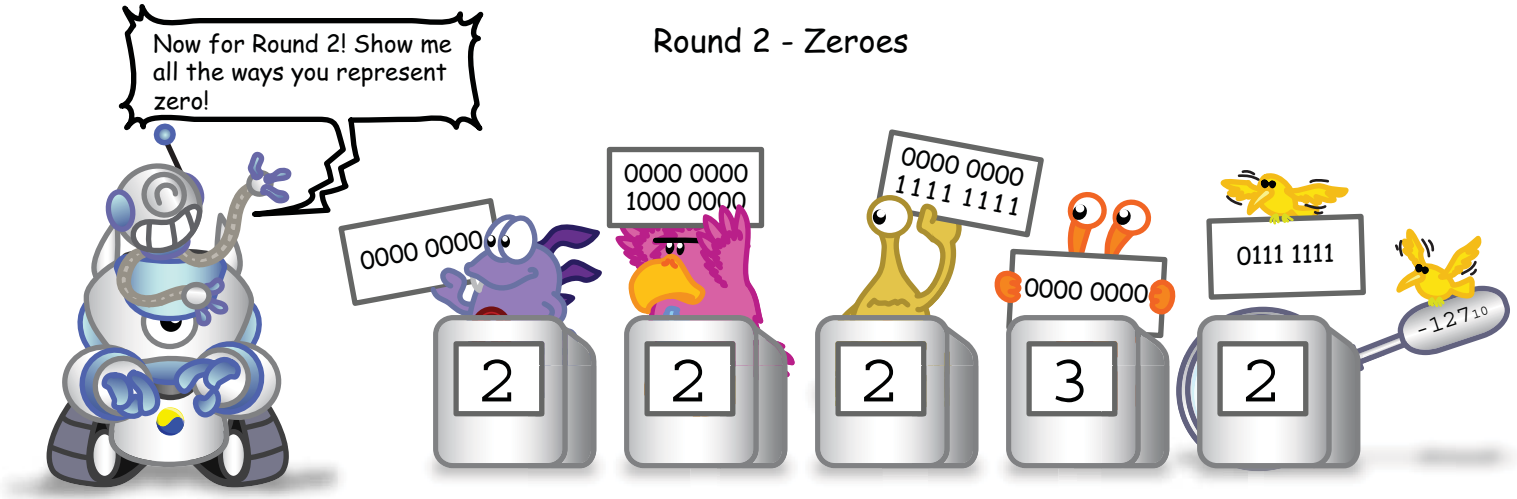
In this competition, we'll use 8-bit numbers. Now let's get started!

Round 1 - Negation



Oh, dear! It looks like Unsigned can't negate. But this competition has only started, so Unsigned still has a chance of catching up to the others.

Round 2 - Zeroes



Now things are getting interesting! Unsigned and Two's Complement get two points each for having one zero and being able to represent zero with all zero bits. Bias's zero isn't all zero bits, but it gets a point for having only one zero. And though they have two zeroes, Sign Magnitude and One's Complement get a point for having a zero of all zero bits.

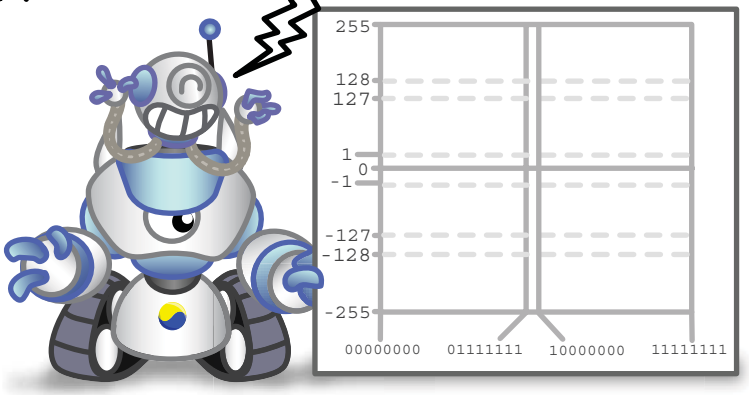
Ketrina Yim

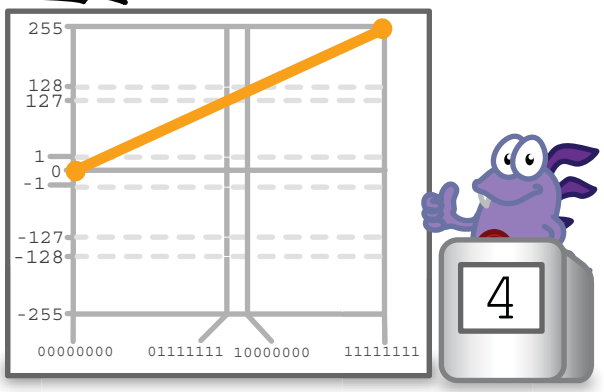
Comparing Integer Representations

Increments and Monotonicity

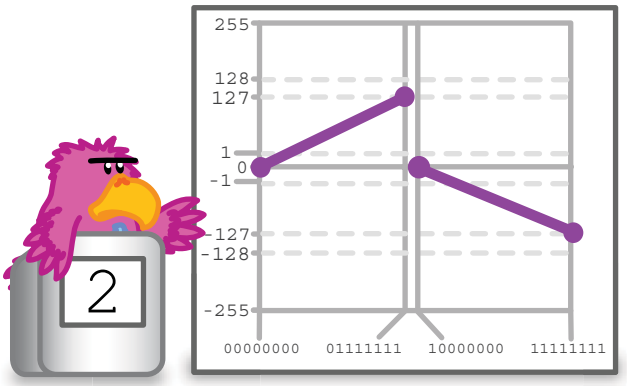
Round 3 - Incrementing

On to Round 3! Using this board, graph how your value changes when you increment your bit pattern from 00000000 to 11111111! We'll give each player a point for having a continuous graph and a point for a consistent unit slope.

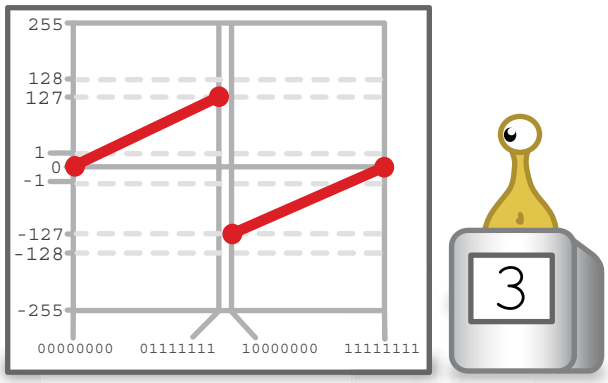




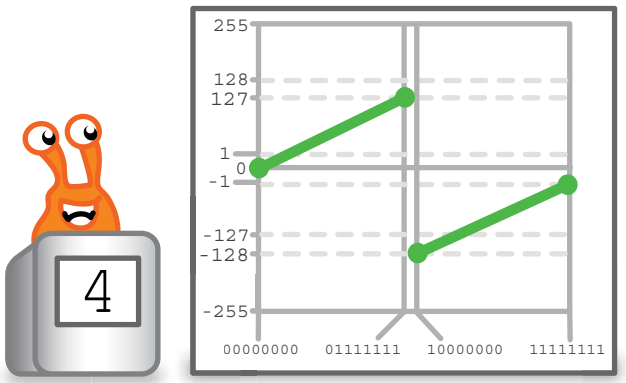
Very nice! Unsigned has a graph that is continuous and has a unit slope. This means we can use an unsigned comparator to compare integers! We'll give Unsigned two points for that.



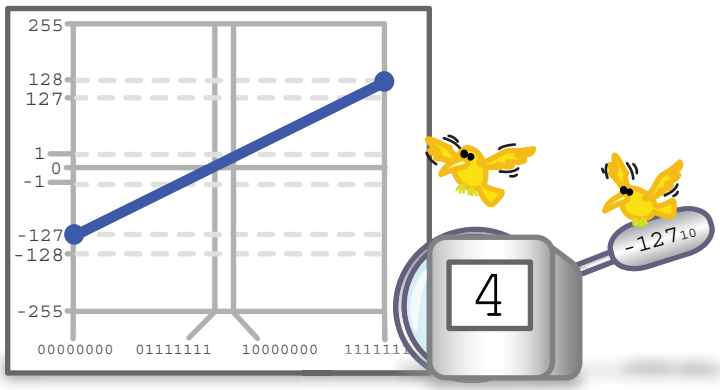
What's this? Sign Magnitude has a very unusual increment indeed. It has a unit slope for positive integers, but the slope becomes -1 for negatives. Sorry, but no points for Sign Magnitude this round.



There's a discontinuity in the graph for One's Complement, but we do like how it has a consistent unit slope. That's one point for One's Complement.



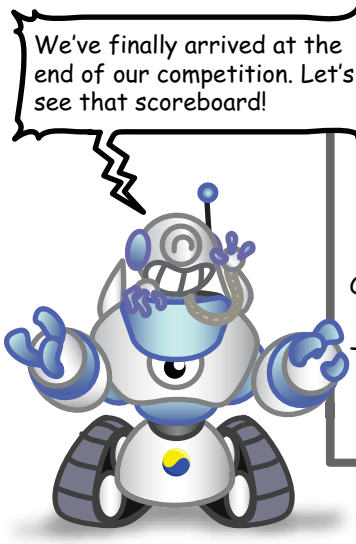
Just like One's Complement, Two's Complement has a discontinuous graph and unit slope. So we'll give Two's Complement a point.



Another monotonically increasing graph with unit slope! You can use an unsigned comparator to compare integers here, too. Bias gets two points!

Comparing Integer Representations

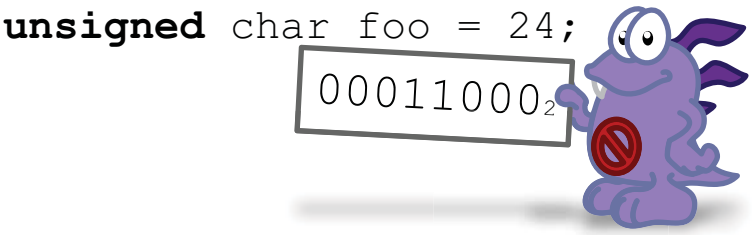
The Thrilling Conclusion!



	Negation?	One Zero?	Zero = 0000 0000	Continuous?	Monotonically Increasing?
Unsigned		✓	✓	✓	✓
Sign Magnitude	✓		✓		
One's Complement	✓		✓		✓
Two's Complement	✓	✓	✓		✓
Bias	✓	✓		✓	✓

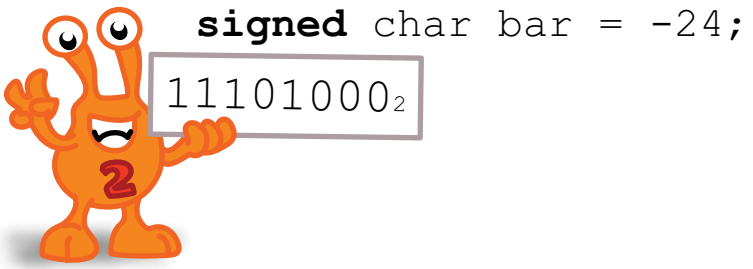
Well, well! It appears we have a three-way tie among Unsigned, Two's Complement, and Bias! We can certainly give each of our winners a prize, though!

Unsigned, you'll be the representation for data whenever users call upon the **unsigned** modifier in C! I've heard that other languages use it, too, so you'll work for them as well.



Bias, you'll represent the exponent in IEEE-754 floating-point numbers! The fact that we can compare exponents with an unsigned comparator will come in handy!

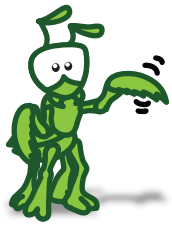
And you, Two's Complement, because you can negate and have one zero that is expressed as all zero bits, you will be the representation of integers for binary computers all around the world!



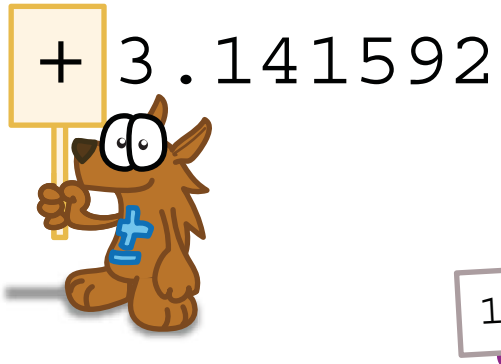
Floating-Point Numbers!

An IEEE 754 floating point number consists of three parts:

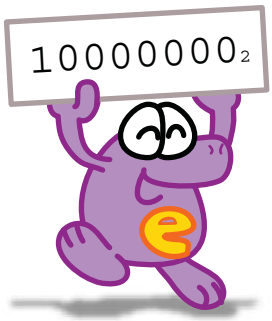
the Sign, the Exponent, and the Mantissa.



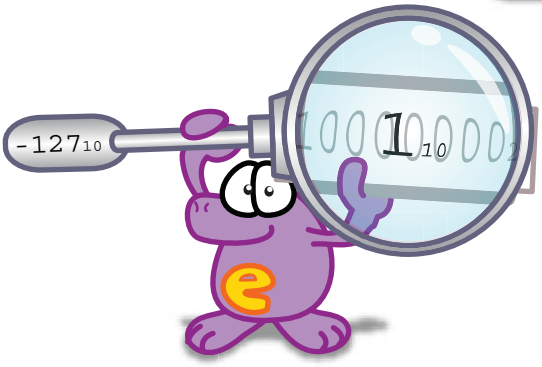
(Also known as the Significand)



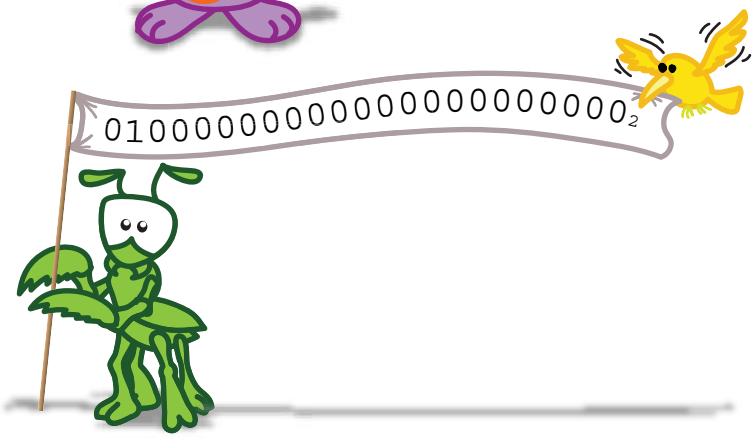
The Sign, as its name suggests, determines the sign of the number.



The Exponent plays a vital role in determining how big (or small) the number is. However, it's encoded so that unsigned comparison can be used to check floating-point numbers.



To see the true magnitude of the Exponent, you'd need to subtract the Bias, a special number determined by the length of the Exponent.

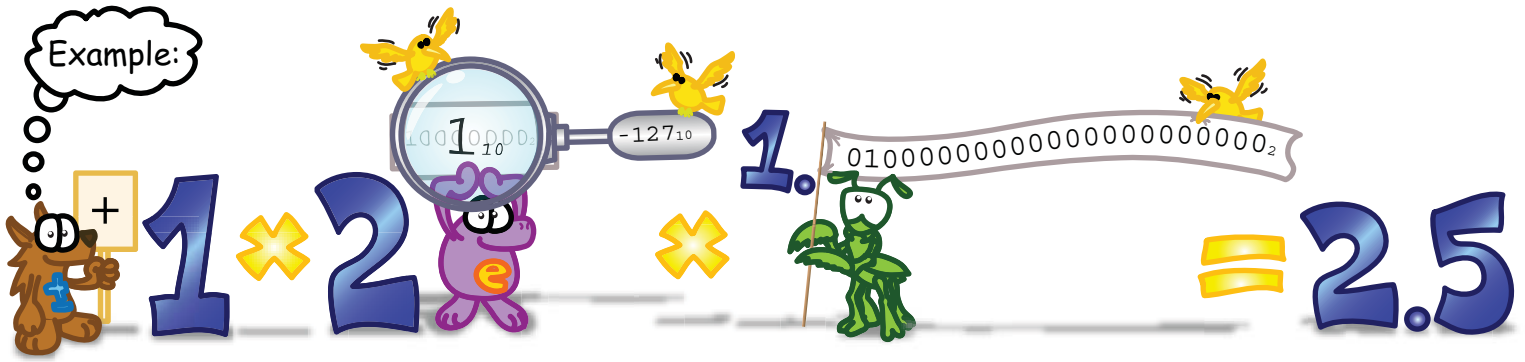


And last but not least, the Mantissa holds the significant digits of the floating point number.

Ketrina Yim

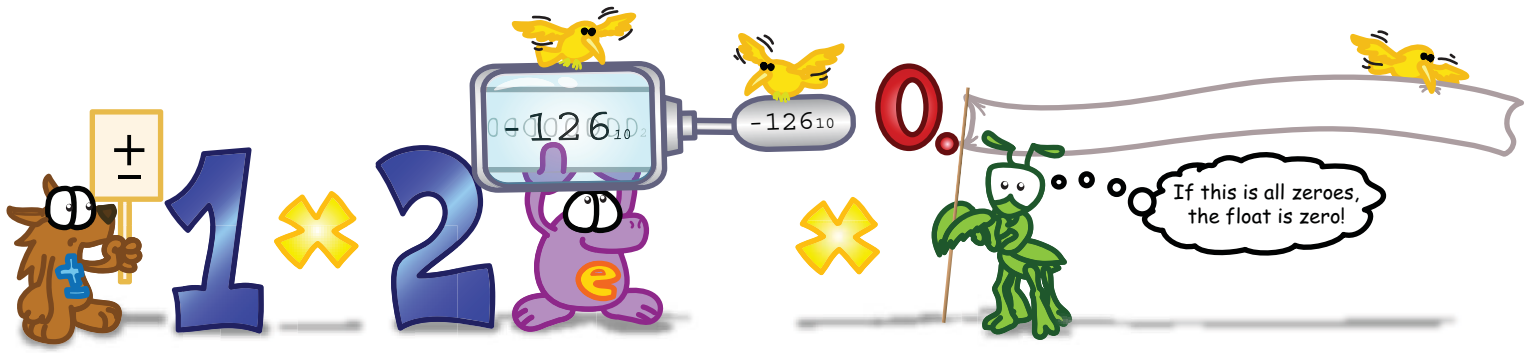
Floating-Point Numbers:
All Together Now!

Once all the parts of the floating-point number are obtained, converting it to decimal is just a matter of applying the following formula:

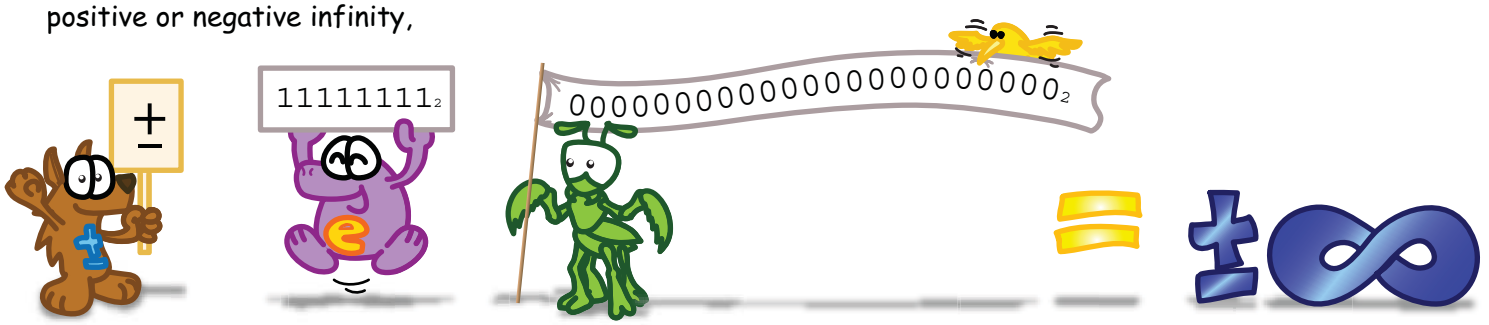


Notice that the Mantissa actually represents a fraction with an implicit 1 in front of it, instead of an integer! In addition to representing real numbers, the IEEE 754 representation can also indicate...

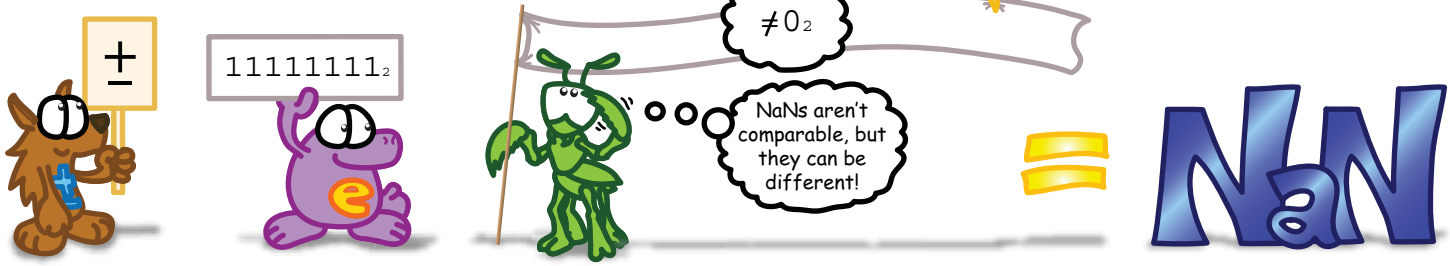
the set of numbers known as denormalized numbers (including zero),



positive or negative infinity,

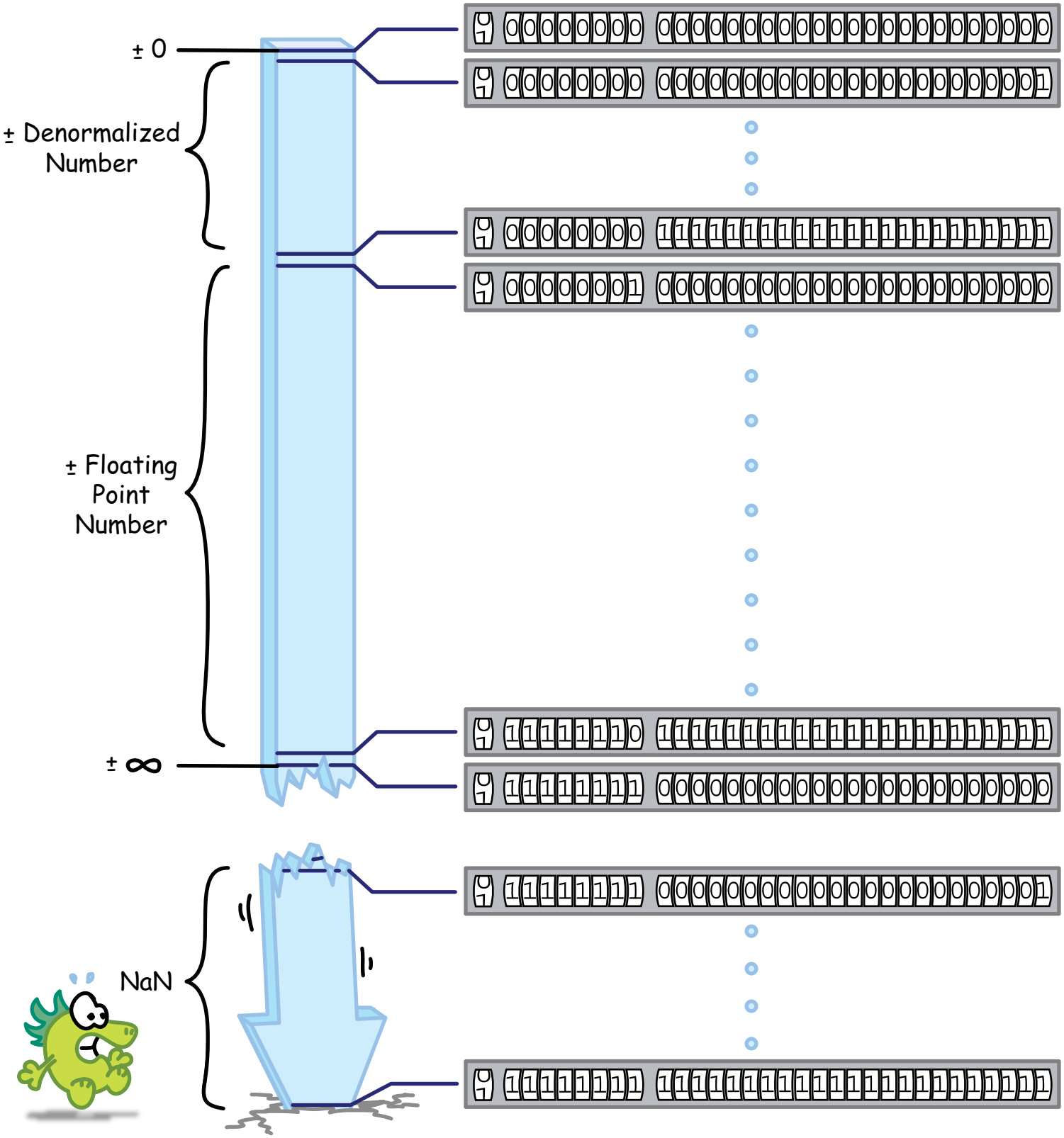


and even when something is not a number! This is called NaN.



Floating-Point Numbers:
The Great Number Line

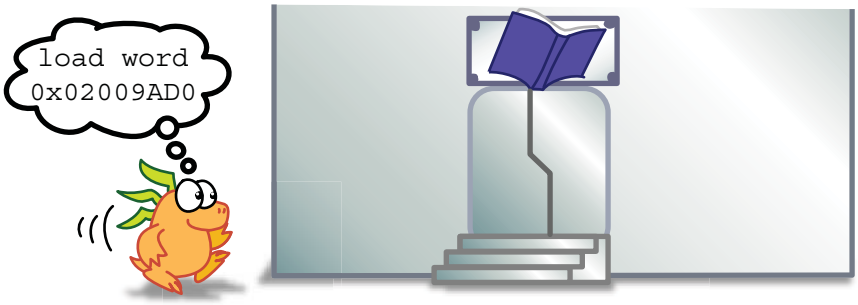
Due to the format of the IEEE-754 standard, the floating-point numbers can be plotted on a number line. In fact, the floating-point numbers are arranged so that they can be incremented like a binary odometer!



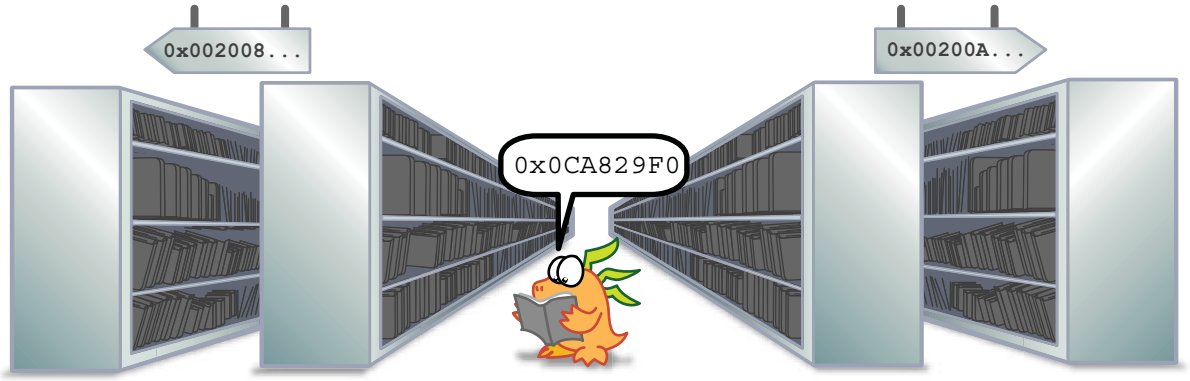
Katrina Yim

Caches: What Are They For?

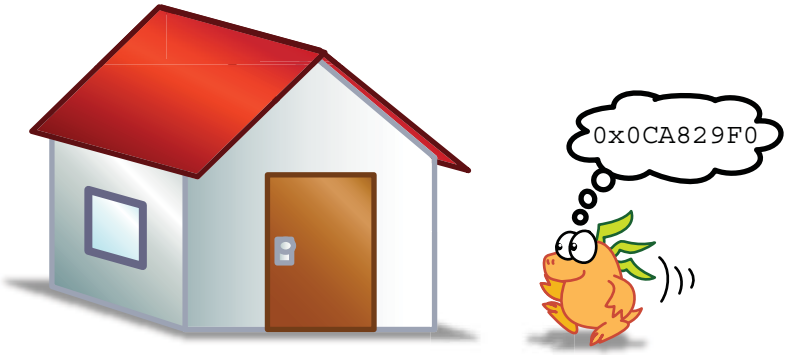
For computers, memory accesses are like going to the library,



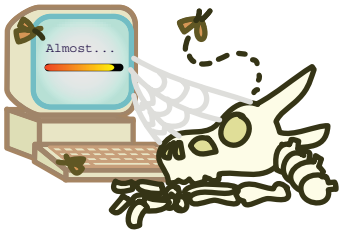
Finding the necessary information in the page of a book,



And going back home to do the work involving that information.



While computers don't mind going back and forth like this for data, it usually means users have to do a lot of waiting.

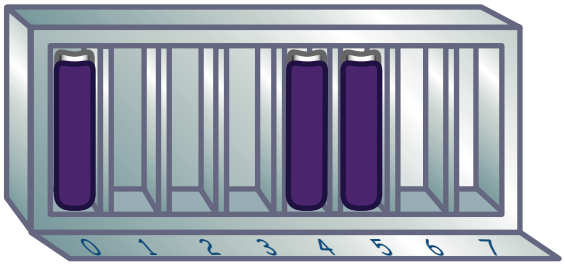


Fortunately for users, computers have caches, which is the equivalent of keeping copies of the books needed on a shelf near the workspace. Through a number of mechanisms, caches give the illusion of being able to access memory very quickly!

Cache Associativity

Just as bookshelves come in different shapes and sizes, caches can also take on a variety of forms and capacities. But no matter how large or small they are, caches fall into one of three categories: direct mapped, n-way set associative, and fully associative.

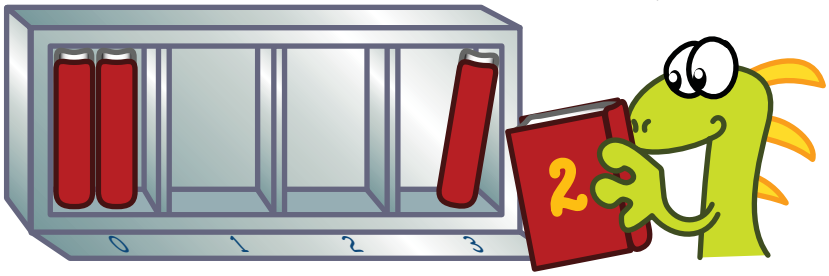
Direct Mapped



Tag	Index	Offset

A cache block can only go in one spot in the cache. It makes a cache block very easy to find, but it's not very flexible about where to put the blocks.

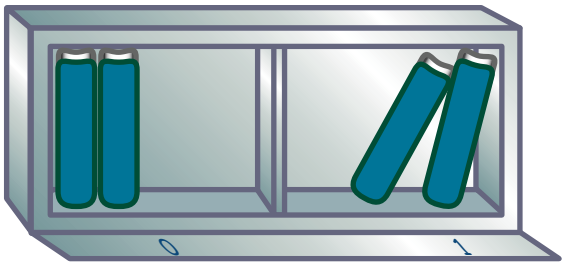
2-Way Set Associative



Tag	Index	Offset

This cache is made up of sets that can fit two blocks each. The index is now used to find the set, and the tag helps find the block within the set.

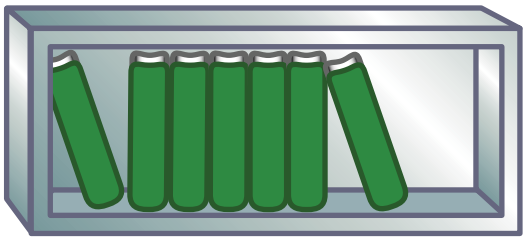
4-Way Set Associative



Tag	Index	Offset

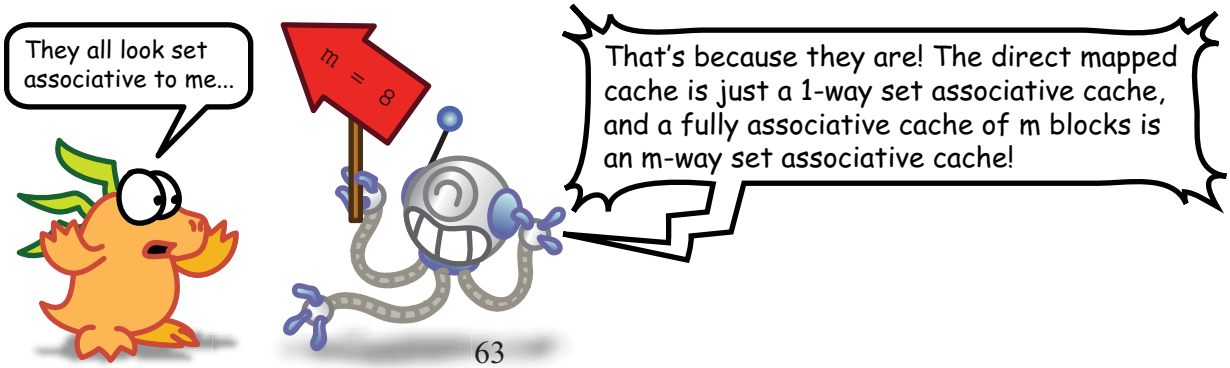
Each set here fits four blocks, so there are fewer sets. As such, fewer index bits are needed.

Fully Associative



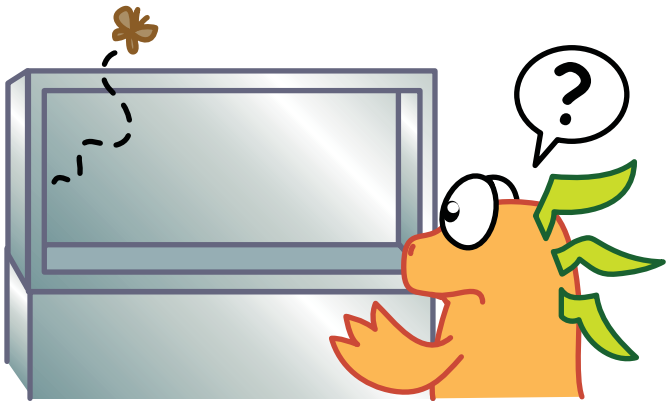
Tag	Offset

No index is needed, since a cache block can go anywhere in the cache. Every tag must be compared when finding a block in the cache, but block placement is very flexible!



Katrina Yim

Sometimes, the cache doesn't have the memory block the computer's looking for. When this happens, it's called a cache miss. There are three causes of cache misses. Just remember the three C's:

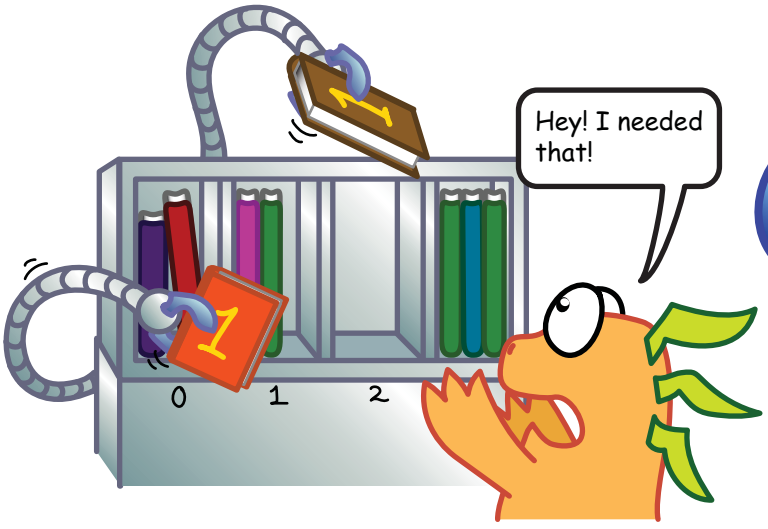


Compulsory

Compulsory misses happen when a block is referenced for the first time. The computer can't get a block that doesn't exist yet!

Capacity

The block is not in the cache because there is no space in the cache for it. Caches are of finite size, after all.

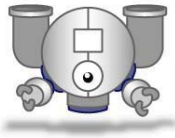
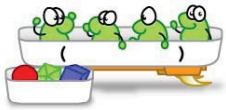
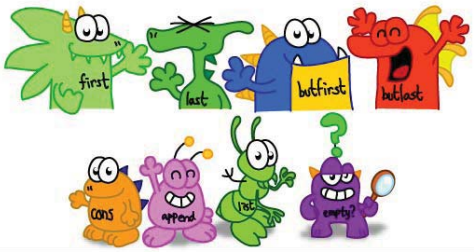
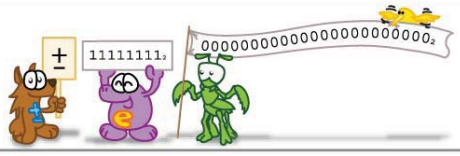
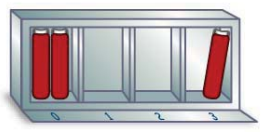
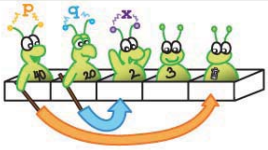


Conflict

These types of misses happen only in direct-mapped and set-associative caches. Multiple blocks can be mapped to a set, forcing evictions when the set is full.

10.6 Appendix F: List of Visual Metaphors

Below is a list of the visual metaphors used in the illustrations so far, along with a written explanation of each metaphor's meaning.

	Scheme functions are represented by machines with input and output pipes. The number of input pipes (pipes pointing upwards) represents the number of arguments the function takes.
	Words and atomic datatypes in Scheme are green characters, except for the empty word. The empty word is invisible, so it can only be seen when it is holding a sign with double quotes on it.
	Scheme sentences and lists are represented by white boats. The shapes symbolize arbitrary elements.
	An assortment of characters are used to describe the inner workings of the Scheme functions. Each character is labeled with the function's name, and their actions visually represent how the function works. The top row contains the word and sentence selectors, while the bottom row has the list constructors and the `empty?` predicate.
	A Scheme data stream is represented by a stream of water, with the values floating in it. In this image, the stream contains key-value pairs, shown as boats with values holding large keys.
	Various methods of binary integer representation, from left to right: bias, unsigned, sign magnitude, one's complement, and two's complement. Each character possesses a feature indicating the type of representation it symbolizes.
	The components of a floating-point number, from left to right: the sign bit, the exponent, and the mantissa. These are presented as characters holding signs.
	Caches are represented by bookshelves. Each segment of the bookshelf is a cache block set, for set-associative or direct-mapped caches. A bookshelf with no dividers is a fully-associative cache.
	Memory is represented by a row of boxes, occupied by values (green characters). If a value belongs to a variable, the variable's name appears above the character's head. The arrows held by some of the values represent pointers.