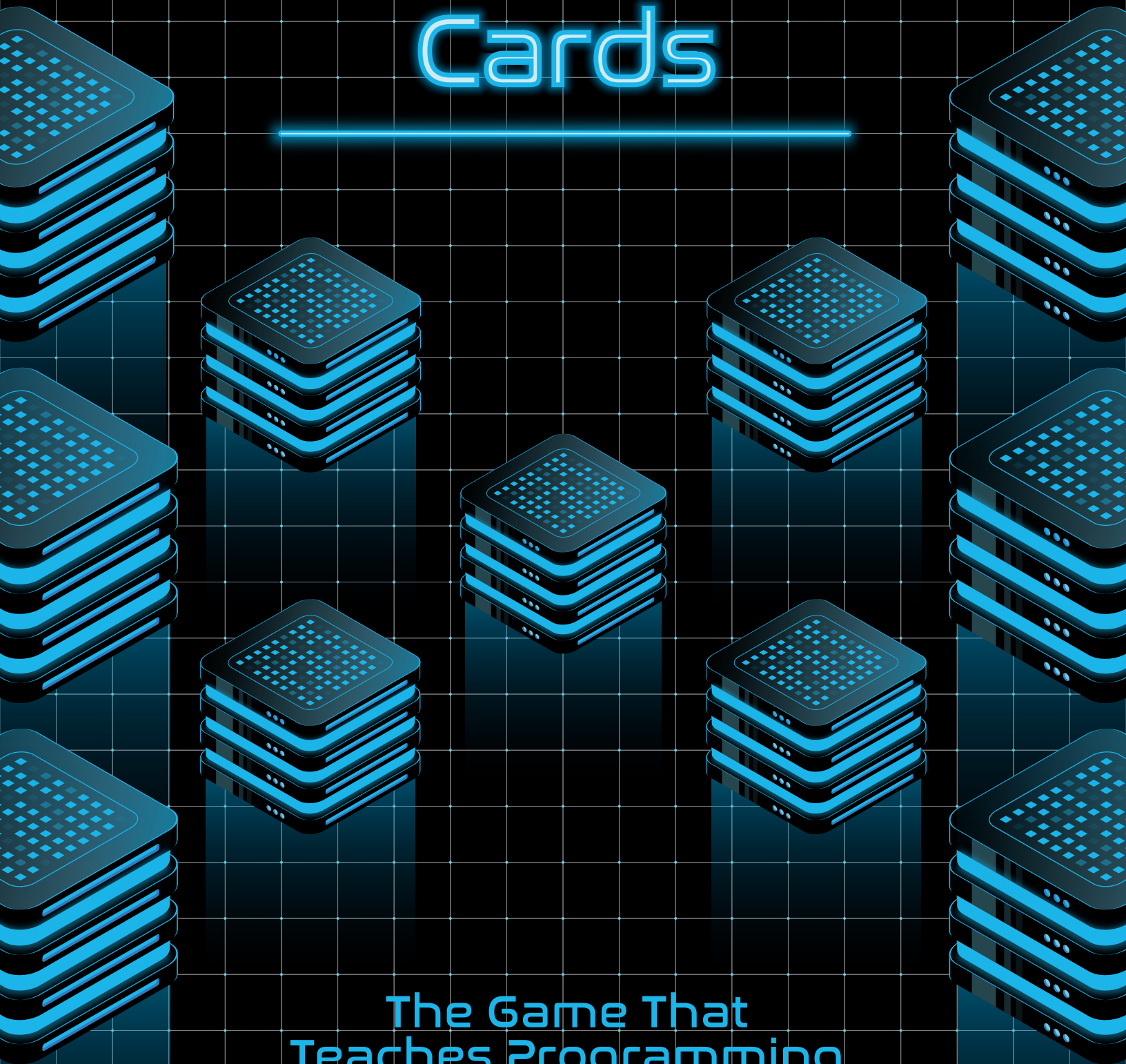

Programming Cards



The Game That
Teaches Programming
by Michael A. Whitehead

Programming Cards

The Game That Teaches Programming

Michael A. Whitehead

Copyright © 2025 – Michael A. Whitehead
All Rights Reserved.
ISBN: 979-8-9911615-0-3

LEGO® is a trademark of the LEGO Group of companies which does not sponsor, authorize
or endorse this book

Contents

Acknowledgments	i
Preface	iii
Introduction	v
1 How to Play the Game	1
2 Types and Basic Functions	11
3 Creating Our Own Functions	57
4 Conditionals	73
5 Understanding Recursion	93
6 Lists	157
7 User Input	227
8 FizzBuzz	235
9 Programming a Car	245
10 A Letter To Future Programmers	251
A Solutions	255

Acknowledgments

Many people deserve my thanks, but none more so than my wife. Without her, none of this would be possible. Thank you, dear wife. I love you.

I also want to acknowledge my four kids, who provide me with the greatest joy and satisfaction in life. Not only were they the guinea pigs, but their faces and voices in the back of my mind motivated me and guided my efforts to completion. Especially thanks to my oldest son, who, as a newborn, refused to sleep through the night unless I was pacing the hallway with him. As I paced the hallway, with him in one arm and a white paper in the other, I learned many things that are uncommon knowledge among most programmers today. Thanks to his encouragement, I initiated a more profound journey into the world of programming.

I owe a lot to the pioneers of computing and the knowledge they shared. Thanks to Simon Peyton Jones. Of all the academics and contributors to programming languages, his work inspired me more than any other. I read mostly his books and research papers during those many sleepless nights.

Lastly, I need to thank two more people: Robert Martin and William Byrd. The card game was inspired from their presentation on teaching concatenative programming using sticky notes.

Preface

Perhaps the real origin of this book was when I walked into my oldest son's first-grade classroom. It was parent-teacher conference and I had a feeling of excitement. I was eager to hear how he was doing from his teacher's perspective. A few minutes into the meeting I realized the teacher was not very impressed with my son and I was a little surprised. I didn't expect my son to be the smartest, but I expected more positive feedback than I received. The main issue the teacher focused on was math. In my head I was already planning how I would work with him. As the conversation continued, however, I realized that there were more significant issues. I asked many questions about his behavior in the classroom, what kind of activities the class did, and which areas my son performed well in. The teacher couched the criticism of my son carefully, but the gist of the situation was that the teacher believed the solution to improve my son's education was at home. The teacher didn't just want help catching him up with math, but more importantly, a stern push to be quiet and obedient. I believe the teacher was wrong about many things, but the teacher was right about one thing. If my children were to have an education it needed to start in my home.

After getting home I had a conversation with my son to understand things from his perspective. He loved school, but he admitted he was too talkative and had trouble paying attention. For the next few weeks we drilled math flash cards. There was enormous improvement in his competency, but that's not the point of this story. The point is that after all that work I started to think more about what my son was actually doing in that school. I would dwell on things his teacher said during that parent-teacher conference. Things that were seemingly insignificant at the time, but were critical to my son's education. I was so focused on my son's behavior and inadequate scores that I failed to realize that almost no time was spent on essential subjects like reading, writing, and arithmetic. Instead, time was spent learning to be quiet, obedient, and how not to stand out. Even the amount of time left for socializing was minimal. After thinking through these issues, my wife and I decided it was time to homeschool.

The decision to homeschool came from many more conversations between my wife and me, but when we finally started we were amazed at how easy it was compared to our expectations. Of course homeschooling is not easy, but we thought it would be much more difficult. One thing that made it easier was that there are so many books to help parents with almost any subject. Homeschooling curriculum is cheap, yet many high-quality options exist to make the subject more fun or interesting than the typical public school experience. However, when it comes to programming I found the material to be lacking.

I would break down traditional education for programming into two categories. The first is what I call the kindergarten environment. The kindergarten environment is where

a website or application dumbed down programming and holds the learner's hand to such a degree that many of the fundamentals are glossed over or skipped entirely. The second category is the journey to Mount Doom. It is an arduous, long journey with many obstacles that will make you question why you even care about programming. Before writing this book my advice to those using a kindergarten environment was that if they want to be programmers, then at some point, they need to start the journey to Mount Doom. My advice to those embarking on the journey to Mount Doom is that they need to expect it to be hard. More specifically, be prepared to sit and read, stand and read, lie and read, then read some more. You must read so much that you should invest in good computer screens and ergonomic chairs. To walk the path to Mount Doom you must spend an unhealthy amount of time in front of a computer, so much so that your face won't look that different from Frodo's at the end of the Lord of the Rings trilogy.

My kids don't want to walk to Mount Doom. Most kids don't. Most people don't, which is why most people are not programmers. I wrote this book to create a third category. I call this category the infinite wish category. It is like when a genie grants you a single wish and you wish for an infinite amount of wishes. In this category, just like you would expect being with a genie to be fun and interesting, this learning experience should be fun and interesting. It cannot be overstated that if we want to encourage lifelong learning in our children it must be fun or interesting. As for the infinite wishes, this comes from taking a liberal arts approach to education. Learning how to learn is more important than learning something specific. To learn everything about programming you may need an infinite amount of wishes, but this book gets you through your first handful of wishes, leaving you with many, many more.

I have enjoyed working through the exercises in this book with my children and love seeing them develop the fundamentals of programming. I now want to share this with the world and I hope whoever reads this gets as much out of the book as my family has.

Introduction

Programming is the art of using logic, creativity, and critical thinking to produce a set of instructions for an automaton to produce a desired result. In the case of this book, we are the automaton or computer. In fact, this is where the original meaning of the word computer came from. It meant a person who computes. Now the term computer refers to a mechanical automaton. Programming is not necessarily about coding. In the liberal arts approach to learning programming we abstract logic and critical thinking from the text editor. Nobody knows what coding will look like in 20 years, but we do know what logic will look like. Today professional programmers perform their job by editing text files on a computer. Tomorrow we could be programming by moving virtual objects around in a virtual reality headset.

There are many ways to abstract programming and represent its most basic form of logic, but this book has the requirement of making it fun and interesting for a young audience with short attention spans. In order to meet this requirement this book takes the approach of programming by physically moving objects. This idea is used in the Montessori school to teach other subjects with amazing results and we are leveraging that idea here.

Much of the training material out there focuses on learning a specific language, like Java, Python or Javascript. These technologies will change in a few years, and today, they are specific to trades within programming. I want my kids to understand the fundamentals. They should be able to quickly pick up any language, and even if they are never programmers, they can still utilize these skills and knowledge to excel in whatever they do.

Who This Book is For

This book is primarily for parents and instructors to guide them in teaching programming to their children and students. This book will equip any parent or teacher, no matter how technical, with the knowledge necessary to teach programming. For your sanity the recommendation is to start teaching kids around eight years old. That is where I've had the most success. Although my six-year old was able to understand the game, it took a good amount of help from me, who designed the game and knows it inside and out. My eight-year-old could grasp the concepts from me by simply reading the book aloud. An understanding of simple arithmetic up to division is helpful but not necessarily required. Because of the interactive way this book covers the material, it can even be adapted to introduce reading, writing, and arithmetic to kindergarten or preschool-aged children with a subset of the concepts. While I wrote this book with young children in mind, there is no upper age limit. This book is also suitable for teaching teenagers and even adults.

A Message to the Teacher or Parent

It is alright if the learner needs help finding the solutions. Many professional programmers can only do their jobs by looking at another programmer's solutions. A word of warning though: Do not use looking up the solution as a crutch! They will only get out of this what they put in. Fortunately, looking at the solution doesn't necessarily spoil the education of programming. If it is necessary to reveal a solution, then don't move on. Have them rework the exercise from scratch even if they remember the solution. The act of physically performing the exercises will help develop understanding and intuition. Also, there are many ways to create a solution. The ones provided in the book are just for guidance.

Some crucial components of a liberal arts education are interdisciplinary learning and effective communication. Look for opportunities to incorporate other subjects into programming or programming into other subjects. For example, you could bring programming into history by looking into the history of computing or bring history into programming by writing a program that outputs a list of famous programmers. Each chapter begins with a quote from a pioneer of programming. I hope you and the learners find their words as inspiring as I have and that you take some time to learn about their contributions.

When it comes to testing, the best way is to have them teach the concept to someone else. Testing for the right solution can be the first step, but there is a reason why a classical liberal arts education requires writing many essays. The essay gauges the student's ability to communicate their understanding. Explaining a concept to someone else solidifies and reveals competency. Look for opportunities for your learners to teach what they have learned to another sibling, student or adult, or even have them teach it right back to you.

Repetition is the mother of all learning and these skills require practice and repetition. The goal is to use this book to craft drills that enable the development of a particular way of thinking. Logic is a skill that can be developed with practice. As I have tested this game on young children, I am confident anyone can learn these skills if they put in the time.

Treating it Like a Game

Children love games even when it isn't really a game. Just calling something a game piques their interest. Go ahead and try it next time you want them to do chores. Say, "Let's play a game! It's called the Clean The House Game!" You may get some eye-rolling, but you had their attention for about two seconds. In all seriousness, improving dull and undesirable situations is possible by making a game out of it. Unfortunately, the inverse is also true and far too ubiquitous in our education system. Things inherently fun, exciting, and playful are turned into monotonous tasks that children reject. Their eyes glaze over and they learn the worst possible thing...that they don't like learning.

Programming is a game. Every program to be written is a puzzle to be solved. I believe it is important to learn the ideas in this book, but I also understand that they can be abstract, philosophical, or mind-bending ideas. Learning abstract ideas by themselves has limited success even among professional programmers. To increase the success rate we need something to bring the ideas from the abstract ether of thought to the concrete physical world. That is why this is a card game you play at the dinner table, not an app you play

in isolation. This book should be treated in the same regard as other popular card games. It should be a fun social event where everyone can solve puzzles together. The difference between this game and others is that in the Programming Cards game everyone who plays wins, and the more you play the more you win.

The goal of this book is that whoever plays the Programming Cards game is rewarded with the knowledge of programming. For some people, each step in acquiring knowledge is enough to make them want to take the next step. Many programmers claim that programming feels like a video game because of a constant challenge and reward cycle. Although, only some enjoy the grind of learning. For those less motivated consider that solving puzzles doesn't have to be the only reward in the game. An example taken from many popular video games is a system of experience points. For every puzzle you solve you gain 10xp. For every new concept learned you gain 20xp. Then, you can use the experience points as a currency to buy things from a marketplace. Some ideas for items at the Programming Cards game marketplace are candy, toys, a special movie night, or a ticket to get out of doing a chore. Remember that we, as parents, understand how knowledge benefits our kids, but they may not have the same level of motivation.

Buying the Cards

This book is about teaching programming using a card game, so you need some cards to follow along. If you somehow acquired this book without the accompanying box set, you may purchase a new one at <https://programmingcardsgame.com>. The box set evolved from many test runs with kids. It exists to simplify the experience. For instance, the cards from the box set are easier to handle than homemade alternatives. This slight difference makes it easier for the kids to focus on the programming aspect of this game.

Creating the Cards

If you do not have a box set and do not want to purchase a new one, you can create your own cards. You can use index cards or cut out squares from pieces of paper. You will need something to write with and something to group the cards. The book says to use a rubber band to group cards. Rubber bands may be good on thicker papers like index cards, but another good option would be to use paper clips. Also, a crucial component included in the box set is an outline for the stacks. I found it is easier for the kids to understand when they have that outline and consistency, so I recommend that if you decide to create your own version of the game that you also include your own outline along with the cards.

Chapter 1

How to Play the Game

Programming is not about typing, it's about thinking.

– Rich Hickey

The Computer

Let's play a game! This game is called the Programming Cards game - and it is just as much a role playing game as it is a card game. There are two roles in the Programming Cards game: The programmer and the computer.

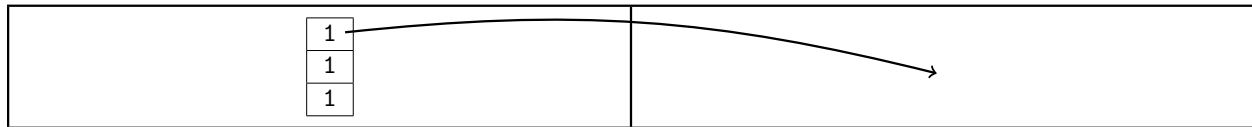
First, let's learn how to act like a computer. The computer is given a stack of cards called the **program**. It is the computer's job to run the program. Each card is a command for the computer to do something. As the computer, you pick up the card at the top of the stack, read it, and execute the command. Execute simply means to do the command. There is another stack called the **data** stack. The results of commands will end up on the **data** stack. When you start a program, only the **program** stack should have cards on it.

Let's make sure we know how to be a computer and run our first program. We are given a **program** stack that looks like the following.

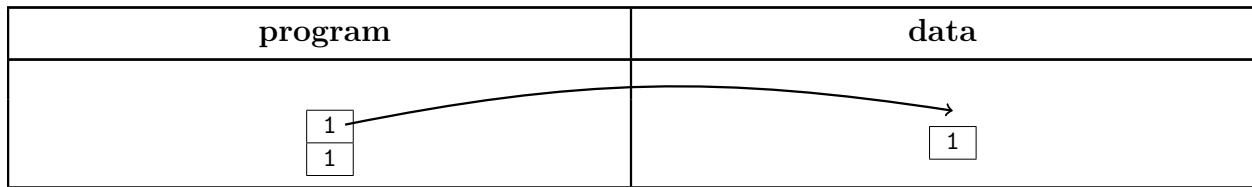
program	
	1
	1
	1

As the computer, we pick up the card at the top of the **program** stack. It is the number **1**. **1** is a number card and a very simple command. The command for number cards is to place them on top of the data stack. Now execute the command, and place the card on top of the **data** stack.

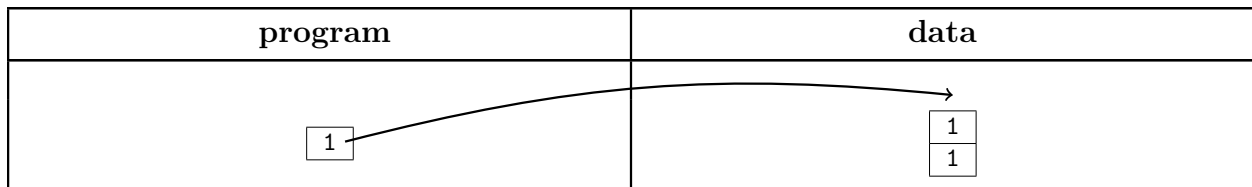
program	data
---------	------



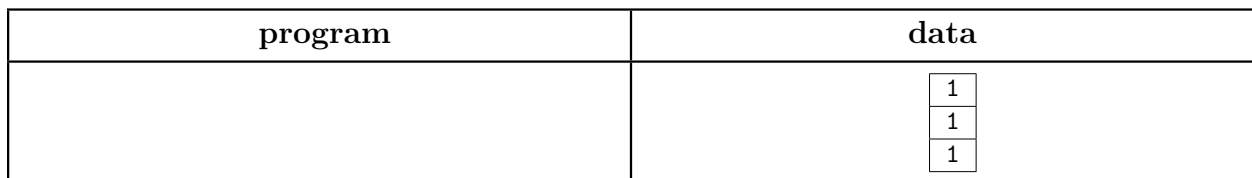
We still have cards in our **program** stack, so our job as a computer is not done. As the computer, we take the card at the top of the **program** stack and execute it. It is also a number card, so execute the number command by moving the card to the top of the **data** stack.



Our **program** stack still has a card, so we still have work to do. Pick up the card, and execute the command.



Good Job! Our **program** stack is empty, and we have no more commands to perform. Our job as a computer is complete. As a computer we don't have a say in the work we do. We just take the list of commands from the programmer and we execute them.



The Programmer

Let's learn how to be a programmer. The programmer's job is to create a list of instructions for a computer. In the Programming Cards game, the list of instructions is a stack of cards. Once we have our list of instructions A.K.A. **program** stack, we hand it off to the computer to run. Write a similar program to the one we just did. Only this time, we want the final result to look like this.

program	data
	2 2 2 2

What should the **program** stack look like? Once you write the program, act like a computer and run it to ensure that you get the right result.

The following table shows the solution and each step for executing the solution. Each row in the table shows what all the stacks should look like at that step. The first row of the table is step 0. This is the beginning of your program when all stacks are cleared except your **program** stack. The second row shows what the stacks look like after taking the first step. The third row shows what the stacks look like after the second step, and so on. These tables are here for you to make sure that you are playing as the computer correctly.

step	program	data
0	2 2 2 2	
1	2 2 2	2
2	2 2	2 2
3	2	2 2 2
4		2 2 2 2

Congratulations! You wrote your first program! Or we could say that you won your first Programming Cards game! This is what it means to play the Programming Cards game. We are given a problem, and we act like a programmer to create a program that will solve that problem. If we create a program that correctly solves the problem then we win!

Teaching the Computer New Commands

As programmers we need more than just number cards to write interesting programs. We need to teach the computer more commands. With only a handful of basic commands a

programmer can combine them to create any program. There are 42 commands in the Programming Cards game. Think about each command as a word. Think about how many words are in human languages like English or Spanish. Compared to human languages, which have thousands of words, programming languages are very small. Learning a human language can take months to years. Learning programming languages can take days to weeks.

The next command we will learn is the $\boxed{+}$ command. Unlike the number cards, the $\boxed{+}$ command has a multi-step process. When you see the $\boxed{+}$ command at the top of the **program** stack, move it to the **proc** stack. Proc is short for process. A card on the **proc** stack tells the computer that it is in the middle of processing a command. The plus command needs two cards. Move two cards, one at a time, from the **data** stack to the **proc-data** stack. Now, the plus command has all the data it needs. Add the two cards on the **proc-data** stack together and replace them with the answer. The last step of the $\boxed{+}$ command is to move the result from the **proc-data** stack to the **data** stack and discard the $\boxed{+}$ card.

Write the program below that includes the $\boxed{+}$ card, then act as the computer to practice the whole process. Use the table as a reference to make sure you are performing the steps correctly.

program	
	1 1 +

Required cards

count	cards
$\boxed{+}$	1
$\boxed{2}$	1
$\boxed{1}$	2

step	program	data	proc	proc-data
0	1 1 +			
1	1 +	1		
2	+	1 1		
3		1 1	+	
4		1	+	1
5			+	1 1
6			+	2
7		2	+	
8		2		

It is important to make sure that you are following the process correctly so don't just speed ahead. Use the tables to double check your processing steps.

Here is another program to get some more practice as the computer. This next program has multiple

+

 commands. As the computer you can only process one command at a time. So when you reach the first

+

 command go through the process entirely until the **proc** and **proc-data** stacks are clear. That means that as you process the first

+

 command you will have two more commands waiting on the **program** stack. Only after a command is completed in it's entirety do you look at the **program** stack to see what the next command is.

program
2 2 + 2 +

Required cards

count	cards
4	1
6	1
+	2
2	3

step	program	data	proc	proc-data
0	2 2 + 2 +			
1	2 + 2 +	2		
2	+ 2 +	2 2		
3	2 +	2 2	+	
4	2 +	2	+	2
5	2 +		+	2 2
6	2 +		+	4
7	2 +	4	+	
8	2 +	4		
9	+	2 4		
10		2 4	+	

11		4	+	2
12			+	4 2
13			+	6
14		6	+	
15		6		

Let's modify the program a little bit. This time the program has all the number commands in the front, and all the $+$ commands in the back. Processing the two $+$ cards at the end will reduce the **data** stack to the same result. Take some time to think about how this works.

program	
	2 2 2 + +

step	program	data	proc	proc-data
0	2 2 2 + +			
1	2 2 + +	2		
2	2 + +	2 2		
3	+ +	2 2 2		
4	+	2 2 2	+	

5	+	2 2	+	2
6	+	2	+	2 2
7	+	2	+	4
8	+	4 2	+	
9	+	4 2		
10		4 2	+	
11		2	+	4
12			+	2 4
13			+	6
14		6	+	
15		6		

The Computer Role Is Important

Learning how to be a computer is a prerequisite for learning how to be a programmer. The better you understand how the computer is going to handle your list of commands the more power you have to craft a list of commands. After learning new commands make sure you have a grasp on the process of that command before moving on. After enough practice, being a computer will become second nature to you.

Playing as the computer should be robotic and almost brainless. The only brain power you need is to know which stacks are which, and the process for each of the 42 commands. Many of the commands have the same process, but there are a few exceptions that we will learn about later.

But you may ask why do I have to be the computer? Why can't I just write the program and hand it off to a real computer like the professional software engineers do? The full answer to this question can be an entire book on its own, but I will give a small explanation as motivation.

Often professional programmers get stuck in the trap of a guess and check style of programming. Many software engineers don't know how to think about what they are trying to achieve, and they use our modern computers as a crutch and ask it over and over if it works now? Many software engineers have had successful careers following that pattern, but they never unleash the full power of programming.

The benefit of the Programming Cards game is that it instills a deeper intuition. It does

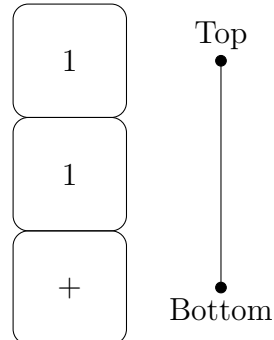
this because *you* are doing the work. There is no way to cheat. When you play as the computer you will see things from a different perspective, and that perspective will help you unleash the full power of programming.

The Programmer Role Is Hard

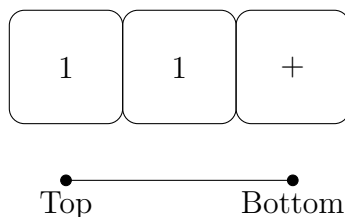
Some people are able to learn programming faster than others, so if you are struggling to understand the puzzles then play with someone who can complete the puzzle. You be the computer and have them be the programmer, and see what you can learn from them. While the Programming Cards game makes learning programming much simpler there is still no getting around the fact that you must learn how to exercise your brain to be a programmer. That doesn't mean that you have to find the answer on your own though.

When acting as the programmer it is better to lay out the **program** stack flat so you can see all the cards at once. This is in contrast to the computer where it is better to work with one process at a time. As the programmer, we want to see as much of the program as possible. Remember that when you lay your cards in a flat stack, you must decide which end is at the top and which end is at the bottom.

Flat Vertical Layout



Flat Horizontal Layout



Once you write your program in a flat style, then you can stack them one on top of the other before giving it to the computer.

Puzzle 1.1

Write a program that adds 1 plus 1 plus 1 plus 1 plus 1. The program should produce the result of the single 5 card on the **data** stack.

Congratulations again! You won another game! Now you can create programs that add numbers. There are even more interesting programs that we can write, and the way that we learn to write them is by learning more commands.

Chapter 2

Types and Basic Functions

If you think it's simple, then you have misunderstood the problem.

– Bjarne Stroustrup

Strings

It's time to learn how to work with words. When we see a card that has something in double quotes, we call that a string. For example, "hello". The command for a card that is a string is the same as it is for numbers. We simply put the string on top of the **data** stack.

step	program	data
0	"world" "hello"	
1	"hello"	"world"
2		"hello" "world"

Now, we can have our computer talk to the world. In the Programming Cards game, we see two kinds of words on the cards: words like + and words in double quotes. The words that are not in double quotes are what we call functions. + is a function that means to add two numbers together. The words in double quotes are what we call string literals. It is literally the word that is in double quotes.

The reason it is called a string goes back to how computers were originally designed to work with words. They don't think about whole words. Instead, they think about one character of the alphabet at a time. A string to a computer looks like a stack of alphabet characters like this

h

e

l

l

o

. In the Programming Cards game we keep it simple and keep our strings together in quotes on a single card like

"hello"

.

Bugs

Sometimes programmers make mistakes. Even the most brilliant programmers will admit to this. You will also make some mistakes as you go through this book. Even though it is sure to happen, we still should learn ways to avoid common mistakes. Let's examine what it means to make a programming mistake so that we can understand how to prevent it.

Let's look at a straightforward addition program.

Required cards

count	cards
+	1
2	1
1	2

step	program	data	proc	proc-data
0	1 1 +			
1	1 +	1		
2	+	1 1		
3		1 1	+	
4		1	+	1
5			+	1 1
6			+	2
7		2	+	
8		2		

This program will add $1 + 1$, resulting in 2. We've done this before. This time, though, we will analyze the program a bit more. There are two types of cards in this program. The number cards and the

+

 card. The number cards are what we call a number type. The

+

 card is what we call a function type. Remember the command for

+

? The command is to add the two numbers on the **proc-data** stack. What should we do if one of those cards isn't a number?

Let's try it.

step	program	data	proc	proc-data
0	1 "hello" +			
1	"hello" +	1		
2	+	"hello" 1		
3		"hello" 1	+	
4		1	+	"hello"
5			+	1 "hello"

Uh oh! How are we supposed to add "hello" and 1 together? We can't! The computer knows there are more instructions to execute, but it doesn't know how to execute them so it crashes. The computer says I don't know what to do and it quits. You might have seen this happen on your actual computer. Sometimes, an application just dies or quits unexpectedly. This occurs because the programmers made a mistake.

Here, we made the mistake on purpose to understand it better. What exactly is this mistake? The mistake that we made here is what we call a type error. That is when a function is given an incompatible type. The

+

 card is for adding two numbers together so the two cards need to be number cards, but instead, we have one number card and one string card. The string card is incompatible, which is a type error. If the instructions given to the computer have incompatible types, then it is a nonsense program and can't be run.

Types

In the Programming Cards game we have simple types. You already know about the number and string types. Let's lay them out visually to ensure we understand the different types. Create a pile of number cards. Then, make a different pile of string cards. These can be just ordinary piles; we will not run them as programs.

numbers

1
2
3
4
5
6
7
8
9
10

strings

"1"
"hello"
"goodbye"
"programmer"
"Alonzo Church"
"Alan Turing"

The above are just examples, but you know you did it correctly if you have one pile where every card is a number (without quotes) and in the other pile, every card is something in double-quotes.

Another necessary type is called a boolean. George Boole was a famous self-taught mathematician. He created a new kind of logic called boolean algebra. That is why we named the boolean type after him. A boolean is either the card `true` or `false`. Notice that it is just a plain word on the card, with no double quotes. Let's make a pile of boolean cards.

booleans

true
false
true
true
false
true

Pop quiz! Is `"true"` a boolean? No, it isn't because it has double quotes around it which makes it a string.

The last type we will learn about is called a function type. The `+` card is a function type. A function type is different from the other three types. The function type has two parts: the input type and the output type.

Function	Input	Output
$\boxed{+}$	(Number, Number)	(Number)

We can read this table as, the $+$ function has a type that takes two numbers as its input and produces one number as its output. We experienced this when we ran the programs with $\boxed{+}$ cards. We took two cards off the **data** stack and replaced them with one card. The *two* cards we take off the **data** stack must be number cards, and the *one* card we put on the **data** stack must be a number card.

Also, notice that a part of the function's type is how many types it takes as input. We said that $+$ takes 2 number types as input, but some functions might take 3 or 4 or even more. We need to know how many cards to pull off the stack and how many cards to put on the stack when the function is done. The function's type tells us this information.

Reverse Polish Notation

Writing our programs where the function comes after the numbers might seem confusing. For example the program $\boxed{1} \boxed{2} \boxed{+}$ has $+$ at the end. We are used to seeing $+$ between two numbers. When the $+$ function is in the middle, like so $1 + 2$, it is called infix notation. It is called infix notation because the function goes *in-between* the numbers. We are taught to use infix notation to do arithmetic but it has some downsides. Let's say we have the problem $2 + 2 * 2$. First, we do $2 + 2$ then we... WAIT! That isn't right! We don't do addition first; we do multiplication first. Why?

We have to multiply first because we must follow the order of operations. Please Excuse My Dear Aunt Sally. That is how we are taught to remember that the order goes Parenthesis, then Exponents, then Multiply or Divide, then Add or Subtract. We need these rules so everyone gets the same answer when solving a math problem. If I follow my own rules and you follow your own rules, we will get different answers. Multiplying first results in 6, but adding first results in 8. What if we want to add before multiplying? Then we have to use parenthesis like this $(2 + 2) * 2$. There is a lot to remember with infix notation.

In the Programming Cards game we don't use infix notation. Instead, we use something called *postfix* notation. It is also called Reverse Polish notation. It is called Postfix notation because the function comes *after* the numbers. One benefit of Reverse Polish notation is that we don't have to teach the computer the complicated order of operations. Even more importantly, it makes it so that we, as programmers, don't make as many mistakes trying to remember where to put parentheses. We never have to use parenthesis to say what operation needs to happen first. In Reverse Polish notation the order of operations is the order in which they appear on the stack. If we want to add before multiplying, we do $2 \ 2 \ + \ 2 \ *$. If we're going to multiply before adding, we do $2 \ 2 \ * \ 2 \ +$. No parentheses are needed.

Commutativity

Some functions are commutative, and some functions are not commutative. Commutativity has been known in mathematics at least as far back as 300bc. When a function is commuta-

tive the cards the function works on can move around without affecting the result. Addition is commutative because $3 + 2$ and $2 + 3$ both equal 5. Or in Reverse Polish notation $3\ 2\ +$ and $2\ 3\ +$ both equal 5. We can switch the 3 and the 2, and the result doesn't change.

Required cards

count	cards
2	1
3	1
+	1
5	1

step	program	data	proc	proc-data
0	3 2 +			
1	2 +	3		
2	+	2 3		
3		2 3	+	
4		3	+	2
5			+	3 2
6			+	5
7		5	+	
8		5		

Now lets switch the

2

 and

3

step	program	data	proc	proc-data
0	2 3 +			
1	3 +	2		
2	+	3 2		
3		3 2	+	
4		2	+	3
5			+	2 3
6			+	5
7		5	+	
8		5		

We get the same answer.

Subtraction is a function that is not commutative. $5 - 3$ is not the same as $3 - 5$. Nor is it commutative in Reverse Polish notation. $5\ 3\ -$ is not the same as $3\ 5\ -$. When a function is not commutative, it just means we have to pay attention to the order in which the function *takes* the cards.

The process for the $-$ command is to take two cards from the **data** stack (one at a time) and move them to the **proc-data** stack. Then subtract the bottom card from the top.

Required cards

count	cards
3	1
5	1
-	1
2	1

step	program	data	proc	proc-data
------	---------	------	------	-----------

0	5 3 —			
1	3 —	5		
2	—	3 5		
3		3 5	—	
4		5	—	3
5			—	5 3
6			—	2
7		2	—	
8		2		

Notice how

5

 was on the top of the **program** stack, and

3

 was second from the top. Our **program** stack is in Reverse Polish notation. That means we are telling the computer to do $5 - 3 = 2$. But! On the **data** stack, the cards are reversed! The order of cards gets reversed when we move them from stack to stack because we move them one at a time. When the two cards are moved from the **data** stack to the **proc-data** stack for processing they are reversed again, and the process is to subtract the bottom from the top.

Here are the types for the two functions we know so far.

Plus +

+

 takes two number cards as input, and adds them together, then moves the result to the **data** stack.

Function	Input	Output	Commutative
+	(Number, Number)	(Number)	Yes

Subtract -

—

 takes two number cards as input, subtracts the second card from the first, then moves the result to the **data** stack.

Function	Input	Output	Commutative
—	(Number, Number)	(Number)	No

Other Basic Functions

When we learn about new functions, it is always helpful to understand the type of the function. Types give us insight into a function and guide the programmer to create nonsense-free programs. Ensure you understand each function's type and what the function does before moving on to the next chapter. Practice these functions and learn the processes associated with each function because these are the building blocks for the rest of the game.

All of the functions in this chapter follow the similar process of the $\boxed{+}$ and $\boxed{-}$ commands. They all take their input from the **data** stack and move it to the **proc-data** stack. Then they process it and move the result to the **data** stack.

Dup

$\boxed{\text{dup}}$ is short for duplicate. It takes one card of any type and clones it. It moves the results to the **data** stack.

Function	Input	Output
$\boxed{\text{dup}}$	(Any)	(Any, Any)

Required cards

cards	count
$\boxed{\text{dup}}$	1
$\boxed{\text{"hello"}}$	2
$\boxed{1}$	2

dup example: 1

step	program	data	proc	proc-data
0	$\boxed{1}$ $\boxed{\text{dup}}$			
1	$\boxed{\text{dup}}$	$\boxed{1}$		
2		$\boxed{1}$	$\boxed{\text{dup}}$	
3			$\boxed{\text{dup}}$	$\boxed{1}$
4			$\boxed{\text{dup}}$	$\boxed{1}$ $\boxed{1}$
5		$\boxed{1}$	$\boxed{\text{dup}}$	$\boxed{1}$

6		1 1	dup	
7		1 1		

dup example: 2

step	program	data	proc	proc-data
0	"hello" dup			
1	dup	"hello"		
2		"hello"	dup	
3			dup	"hello"
4			dup	"hello" "hello"
5		"hello"	dup	"hello"
6		"hello" "hello"	dup	
7		"hello" "hello"		

Swap

`swap` takes two cards and swaps their order top card and swaps it with the second from the top card. It moves the results to the **data** stack

Function	Input	Output	Commutative
<code>swap</code>	(Any, Any)	(Any, Any)	No

Required cards

cards	count
2	1
1	1
swap	2

swap example: 1

step	program	data	proc	proc-data
0	1 2 swap			
1	2 swap	1		
2	swap	2 1		
3		2 1	swap	
4		1	swap	2
5			swap	1 2
6			swap	2 1
7		2	swap	1
8		1 2	swap	
9		1 2		

swap example: 2

step	program	data	proc	proc-data
0	1 2 swap swap			
1	2 swap swap	1		
2	swap swap	2 1		

3	swap	21	swap	
4	swap	1	swap	2
5	swap		swap	12
6	swap		swap	21
7	swap	2	swap	1
8	swap	12	swap	
9	swap	12		
10		12	swap	
11		2	swap	1
12			swap	21
13			swap	12
14		1	swap	2
15		21	swap	
16		21		

Rollup

`rollup` takes three cards, moves the third card to the top, and moves the results to the data stack.

Function	Input	Output	Commutative
<code>rollup</code>	(Any, Any, Any)	(Any, Any, Any)	No

Required cards

cards	count
-------	-------

rollup	1
3	1
1	1
2	1

rollup example: 1

step	program	data	proc	proc-data
0	1 2 3 rollup			
1	2 3 rollup	1		
2	3 rollup	2 1		
3	rollup	3 2 1		
4		3 2 1	rollup	
5		2 1	rollup	3
6		1	rollup	2 3
7			rollup	1 2 3
8			rollup	3 1 2
9		3	rollup	1 2
10		1 3	rollup	2

11		2 1 3	rollup	
12		2 1 3		

rollup example: 2

step	program	data	proc	proc-data
0	3 2 1 rollup			
1	2 1 rollup	3		
2	1 rollup	2 3		
3	rollup	1 2 3		
4		1 2 3	rollup	
5		2 3	rollup	1
6		3	rollup	2 1
7			rollup	3 2 1
8			rollup	1 3 2
9		1	rollup	3 2
10		3 1	rollup	2

11		2 3 1	rollup	
12		2 3 1		

Rotate

rotate takes three cards and swaps the top and third cards. The second card stays as the second card. It moves the results to the **data** stack.

Function	Input	Output	Commutative
rotate	(Any, Any, Any)	(Any, Any, Any)	No

Required cards

cards	count
3	1
2	1
1	1
rotate	2

rotate example: 1

step	program	data	proc	proc-data
0	1 2 3 rotate			
1	2 3 rotate	1		
2	3 rotate	2 1		
3	rotate	3 2 1		

4		3 2 1	rotate	
5		2 1	rotate	3
6		1	rotate	2 3
7			rotate	1 2 3
8			rotate	3 2 1
9		3	rotate	2 1
10		2 3	rotate	1
11		1 2 3	rotate	
12		1 2 3		

rotate example: 2

step	program	data	proc	proc-data
0	1 2 3 rotate rotate			
1	2 3 rotate rotate	1		
2	3 rotate rotate	2 1		

3	rotate rotate	3 2 1		
4	rotate	3 2 1	rotate	
5	rotate	2 1	rotate	3
6	rotate	1	rotate	2 3
7	rotate		rotate	1 2 3
8	rotate		rotate	3 2 1
9	rotate	3	rotate	2 1
10	rotate	2 3	rotate	1
11	rotate	1 2 3	rotate	
12	rotate	1 2 3		
13		1 2 3	rotate	
14		2 3	rotate	1
15		3	rotate	2 1
16			rotate	3 2 1
17			rotate	1 2 3

18		1	rotate	2 3
19		2 1	rotate	3
20		3 2 1	rotate	
21		3 2 1		

Pop

`pop` takes one card of any type and discards it.

Function	Input	Output
<code>pop</code>	(Any)	()

Required cards

cards	count
1	1
<code>pop</code>	1

pop example: 1

step	program	data	proc	proc-data
0	1 pop			
1	pop	1		
2		1	pop	
3			pop	1
4			pop	
5				

Multiply *

$\boxed{*}$ takes two number cards, multiplies them together, and places the result on top of the **data** stack.

Function	Input	Output	Commutative
$\boxed{*}$	(Number, Number)	(Number)	Yes

Required cards

cards	count
$\boxed{4}$	1
$\boxed{1}$	1
$\boxed{*}$	1
$\boxed{2}$	2

* example: 1

step	program	data	proc	proc-data
0	$\begin{array}{c} \boxed{2} \\ \boxed{1} \\ \boxed{*} \end{array}$			
1	$\begin{array}{c} \boxed{1} \\ \boxed{*} \end{array}$	$\boxed{2}$		
2	$\boxed{*}$	$\begin{array}{c} \boxed{1} \\ \boxed{2} \end{array}$		
3		$\begin{array}{c} \boxed{1} \\ \boxed{2} \end{array}$	$\boxed{*}$	
4		$\boxed{2}$	$\boxed{*}$	$\boxed{1}$
5			$\boxed{*}$	$\begin{array}{c} \boxed{2} \\ \boxed{1} \end{array}$
6			$\boxed{*}$	$\boxed{2}$
7		$\boxed{2}$	$\boxed{*}$	
8		$\boxed{2}$		

* example: 2

step	program	data	proc	proc-data
0	2 2 *			
1	2 *	2		
2	*	2 2		
3		2 2	*	
4		2	*	2
5			*	2 2
6			*	4
7		4	*	
8		4		

Greater Than >

> takes two number cards and asks if the first card is greater than the second card. It moves the result to the **data** stack.

Function	Input	Output	Commutative
>	(Number, Number)	(Boolean)	No

Required cards

cards	count
false	1
2	1
>	1
true	1
1	2

> example: 1

step	program	data	proc	proc-data
0	2 1 >			
1	1 >	2		
2	>	1 2		
3		1 2	>	
4		2	>	1
5			>	2 1
6			>	true
7		true	>	
8		true		

> example: 2

step	program	data	proc	proc-data
0	1 2 >			
1	2 >	1		
2	>	2 1		
3		2 1	>	
4		1	>	2
5			>	1 2
6			>	false

7		false	>	
8		false		

> example: 3

step	program	data	proc	proc-data
0	1 1 >			
1	1 >	1		
2	>	1 1		
3		1 1	>	
4		1	>	1
5			>	1 1
6			>	false
7		false	>	
8		false		

Less Than <

< takes two number cards asks if the first card is less than the second card. It moves the result to the **data** stack.

Function	Input	Output	Commutative
<	(Number, Number)	(Boolean)	No

Required cards

cards	count
true	1
2	1

<	1
false	1
1	2

< example: 1

step	program	data	proc	proc-data
0	2 1 <			
1	1 <	2		
2	<	1 2		
3		1 2	<	
4		2	<	1
5			<	2 1
6			<	false
7		false	<	
8		false		

< example: 2

step	program	data	proc	proc-data
0	1 2 <			
1	2 <	1		
2	<	2 1		

3		2 1	<	
4		1	<	2
5			<	1 2
6			<	true
7		true	<	
8		true		

< example: 3

step	program	data	proc	proc-data
0	1 1 <			
1	1 <	1		
2	<	1 1		
3		1 1	<	
4		1	<	1
5			<	1 1
6			<	false
7		false	<	
8		false		

Greater than or equal $\geq$

$\geq$ takes two number cards, asks if the first card is greater than or equal to the second card. It moves the result to the **data** stack.

Function	Input	Output	Commutative
$\geq$	(Number, Number)	(Boolean)	No

Required cards

cards	count
false	1
2	1
$\geq$	1
true	1
1	2

 $\geq$ example: 1

step	program	data	proc	proc-data
0	2 1 $\geq$			
1	1 $\geq$	2		
2	$\geq$	1 2		
3		1 2	$\geq$	
4		2	$\geq$	1
5			$\geq$	2 1
6			$\geq$	true
7		true	$\geq$	
8		true		

 $\geq$ example: 2

step	program	data	proc	proc-data
0	1 2 $\geq$			

1	2 ≥	1		
2	≥	2 1		
3		2 1	≥	
4		1	≥	2
5			≥	1 2
6			≥	false
7		false	≥	
8		false		

≥ example: 3

step	program	data	proc	proc-data
0	1 1 ≥			
1	1 ≥	1		
2	≥	1 1		
3		1 1	≥	
4		1	≥	1
5			≥	1 1
6			≥	true
7		true	≥	
8		true		

Less than or equal ≤

$\leq$ takes two number cards, asks if the first card is less than or equal to the second. It moves the result to the **data** stack.

Function	Input	Output	Commutative
$\leq$	(Number, Number)	(Boolean)	No

Required cards

cards	count
true	1
2	1
$\leq$	1
false	1
1	2

$\leq$ example: 1

step	program	data	proc	proc-data
0	2 1 $\leq$			
1	1 $\leq$	2		
2	$\leq$	1 2		
3		1 2	$\leq$	
4		2	$\leq$	1
5			$\leq$	2 1
6			$\leq$	false
7		false	$\leq$	
8		false		

$\leq$ example: 2

step	program	data	proc	proc-data
0	1 2 ≤			
1	2 ≤	1		
2	≤	2 1		
3		2 1	≤	
4		1	≤	2
5			≤	1 2
6			≤	true
7		true	≤	
8		true		

≤ example: 3

step	program	data	proc	proc-data
0	1 1 ≤			
1	1 ≤	1		
2	≤	1 1		
3		1 1	≤	
4		1	≤	1
5			≤	1 1
6			≤	true
7		true	≤	

8		true		
---	--	------	--	--

Is Equal $\equiv$

$\equiv$ takes two cards of any type. This function compares the two cards, and if they are the same, it outputs true. If they are different, it outputs false.

Function	Input	Output	Commutative
$\equiv$	(Any, Any)	(Boolean)	Yes

Required cards

cards	count
"goodbye"	1
2	1
$\equiv$	1
"hello"	2
false	2
1	2
true	2

$\equiv$ example: 1

step	program	data	proc	proc-data
0	1 1 $\equiv$			
1	1 $\equiv$	1		
2	$\equiv$	1 1		
3		1 1	$\equiv$	
4		1	$\equiv$	1

5			$\equiv$	1 1
6			$\equiv$	true
7		true	$\equiv$	
8		true		

$\equiv$ example: 2

step	program	data	proc	proc-data
0	1 2 $\equiv$			
1	2 $\equiv$	1		
2	$\equiv$	2 1		
3		2 1	$\equiv$	
4		1	$\equiv$	2
5			$\equiv$	1 2
6			$\equiv$	false
7		false	$\equiv$	
8		false		

$\equiv$ example: 3

step	program	data	proc	proc-data
0	"hello" "goodbye" $\equiv$			
1	"goodbye" $\equiv$	"hello"		

2	≡	"goodbye" "hello"		
3		"goodbye" "hello"	≡	
4		"hello"	≡	"goodbye"
5			≡	"hello" "goodbye"
6			≡	false
7		false	≡	
8		false		

≡ example: 4

step	program	data	proc	proc-data
0	"hello" "hello" ≡			
1	"hello" ≡	"hello"		
2	≡	"hello" "hello"		
3		"hello" "hello"	≡	
4		"hello"	≡	"hello"
5			≡	"hello" "hello"
6			≡	true
7		true	≡	
8		true		

≡ example: 5

step	program	data	proc	proc-data
------	---------	------	------	-----------

0	true true ≡			
1	true ≡	true		
2	≡	true true		
3		true true	≡	
4		true	≡	true
5			≡	true true
6			≡	true
7		true	≡	
8		true		

≡ example: 6

step	program	data	proc	proc-data
0	false false ≡			
1	false ≡	false		
2	≡	false false		
3		false false	≡	
4		false	≡	false
5			≡	false false
6			≡	true
7		true	≡	
8		true		

≡ example: 7

step	program	data	proc	proc-data			
0	<table><tr><td>1</td></tr><tr><td>"hello"</td></tr><tr><td>≡</td></tr></table>	1	"hello"	≡			
1							
"hello"							
≡							
1	<table><tr><td>"hello"</td></tr><tr><td>≡</td></tr></table>	"hello"	≡	<table><tr><td>1</td></tr></table>	1		
"hello"							
≡							
1							
2	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>"hello"</td></tr><tr><td>1</td></tr></table>	"hello"	1		
≡							
"hello"							
1							
3		<table><tr><td>"hello"</td></tr><tr><td>1</td></tr></table>	"hello"	1	<table><tr><td>≡</td></tr></table>	≡	
"hello"							
1							
≡							
4		<table><tr><td>1</td></tr></table>	1	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>"hello"</td></tr></table>	"hello"
1							
≡							
"hello"							
5			<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>1</td></tr><tr><td>"hello"</td></tr></table>	1	"hello"
≡							
1							
"hello"							
6			<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>false</td></tr></table>	false	
≡							
false							
7		<table><tr><td>false</td></tr></table>	false	<table><tr><td>≡</td></tr></table>	≡		
false							
≡							
8		<table><tr><td>false</td></tr></table>	false				
false							

Is Not Equal ≠

≠ takes two cards of any type. If the cards are not equal, then the output is true; if they are equal, the output is false. It moves the result to the **data** stack.

Function	Input	Output	Commutative
≠	(Any, Any)	(Boolean)	Yes

Required cards

cards	count
"goodbye"	1
2	1
≠	1
false	1
"hello"	2

true	2
1	2

≠ example: 1

step	program	data	proc	proc-data
0	1 1 ≠			
1	1 ≠	1		
2	≠	1 1		
3		1 1	≠	
4		1	≠	1
5			≠	1 1
6			≠	false
7		false	≠	
8		false		

≠ example: 2

step	program	data	proc	proc-data
0	1 2 ≠			
1	2 ≠	1		
2	≠	2 1		
3		2 1	≠	

4		1	≠	2
5			≠	1 2
6			≠	true
7		true	≠	
8		true		

≠ example: 3

step	program	data	proc	proc-data
0	"hello" "goodbye" ≠			
1	"goodbye" ≠	"hello"		
2	≠	"goodbye" "hello"		
3		"goodbye" "hello"	≠	
4		"hello"	≠	"goodbye"
5			≠	"hello" "goodbye"
6			≠	true
7		true	≠	
8		true		

≠ example: 4

step	program	data	proc	proc-data
0	"hello" "hello" ≠			
1	"hello" ≠	"hello"		

2	$\neq$	"hello" "hello"		
3		"hello" "hello"	$\neq$	
4		"hello"	$\neq$	"hello"
5			$\neq$	"hello" "hello"
6			$\neq$	false
7		false	$\neq$	
8		false		

$\neq$ example: 5

step	program	data	proc	proc-data
0	true true $\neq$			
1	true $\neq$	true		
2	$\neq$	true true		
3		true true	$\neq$	
4		true	$\neq$	true
5			$\neq$	true true
6			$\neq$	false
7		false	$\neq$	
8		false		

$\neq$ example: 6

step	program	data	proc	proc-data
------	---------	------	------	-----------

0	true false ≠			
1	false ≠	true		
2	≠	false true		
3		false true	≠	
4		true	≠	false
5			≠	true false
6			≠	true
7		true	≠	
8		true		

Boolean Logic

Booleans are essential building blocks in programming. A boolean is used to create choices in our program. It is crucial to be familiar with functions that work with booleans.

And $\wedge$

$\wedge$ takes two boolean cards and if they are both true will output true, otherwise the output will be false. It moves the result to the **data** stack.

Function	Input	Output	Commutative
$\wedge$	(Boolean, Boolean)	(Boolean)	Yes

Required cards

cards	count
$\wedge$	1
false	2
true	2

$\wedge$ example: 1

step	program	data	proc	proc-data
0	true true $\wedge$			
1	true $\wedge$	true		
2	$\wedge$	true true		
3		true true	$\wedge$	
4		true	$\wedge$	true
5			$\wedge$	true true
6			$\wedge$	true
7		true	$\wedge$	
8		true		

 $\wedge$ example: 2

step	program	data	proc	proc-data
0	true false $\wedge$			
1	false $\wedge$	true		
2	$\wedge$	false true		
3		false true	$\wedge$	
4		true	$\wedge$	false
5			$\wedge$	true false
6			$\wedge$	false

7		false	$\wedge$	
8		false		

$\wedge$ example: 3

step	program	data	proc	proc-data
0	false true $\wedge$			
1	true $\wedge$	false		
2	$\wedge$	true false		
3		true false	$\wedge$	
4		false	$\wedge$	true
5			$\wedge$	false true
6			$\wedge$	false
7		false	$\wedge$	
8		false		

$\wedge$ example: 4

step	program	data	proc	proc-data
0	false false $\wedge$			
1	false $\wedge$	false		
2	$\wedge$	false false		
3		false false	$\wedge$	

4		false	$\wedge$	false
5			$\wedge$	false false
6			$\wedge$	false
7		false	$\wedge$	
8		false		

Or $\vee$

$\vee$ takes two booleans; if either is true, then it outputs true. The output will be false only when both inputs are false. It moves the result to the **data** stack.

Function	Input	Output	Commutative
$\vee$	(Boolean, Boolean)	(Boolean)	Yes

Required cards

cards	count
$\vee$	1
false	2
true	2

$\vee$ example: 1

step	program	data	proc	proc-data
0	true true $\vee$			
1	true $\vee$	true		
2	$\vee$	true true		
3		true true	$\vee$	
4		true	$\vee$	true

5			∨	true true
6			∨	true
7		true	∨	
8		true		

∨ example: 2

step	program	data	proc	proc-data
0	true false ∨			
1	false ∨	true		
2	∨	false true		
3		false true	∨	
4		true	∨	false
5			∨	true false
6			∨	true
7		true	∨	
8		true		

∨ example: 3

step	program	data	proc	proc-data
0	false true ∨			
1	true ∨	false		

2	∨	true false		
3		true false	∨	
4		false	∨	true
5			∨	false true
6			∨	true
7		true	∨	
8		true		

∨ example: 4

step	program	data	proc	proc-data
0	false false ∨			
1	false ∨	false		
2	∨	false false		
3		false false	∨	
4		false	∨	false
5			∨	false false
6			∨	false
7		false	∨	
8		false		

Exclusive Or $\vee$

$\vee$ is called exclusive or. It takes two booleans and only outputs true when one input is true and the other is false. If the inputs are both true or both false, then the output is false. It moves the result to the **data** stack.

Function	Input	Output	Commutative
∨	(Boolean, Boolean)	(Boolean)	Yes

Required cards

cards	count
∨	1
true	2
false	2

∨ example: 4

step	program	data	proc	proc-data
0	true true ∨			
1	true ∨	true		
2	∨	true true		
3		true true	∨	
4		true	∨	true
5			∨	true true
6			∨	false
7		false	∨	
8		false		

∨ example: 4

step	program	data	proc	proc-data
------	---------	------	------	-----------

0	true false $\vee$			
1	false $\vee$	true		
2	$\vee$	false true		
3		false true	$\vee$	
4		true	$\vee$	false
5			$\vee$	true false
6			$\vee$	true
7		true	$\vee$	
8		true		

$\vee$ example: 4

step	program	data	proc	proc-data
0	false true $\vee$			
1	true $\vee$	false		
2	$\vee$	true false		
3		true false	$\vee$	
4		false	$\vee$	true
5			$\vee$	false true
6			$\vee$	true
7		true	$\vee$	
8		true		

∨ example: 4

step	program	data	proc	proc-data
0	falsefalse∨			
1	false∨	false		
2	∨	falsefalse		
3		falsefalse	∨	
4		false	∨	false
5			∨	falsefalse
6			∨	false
7		false	∨	
8		false		

Not ¬

¬ is used to reverse a boolean. It takes one boolean and outputs the opposite of whatever the input was. It turns true into false, and it turns false into true. It moves the result to the **data** stack.

Function	Input	Output
¬	(Boolean)	(Boolean)

Required cards

cards	count
true	1
¬	1
false	1

$\neg$ example: 1

step	program	data	proc	proc-data
0	true $\neg$			
1	$\neg$	true		
2		true	$\neg$	
3			$\neg$	true
4			$\neg$	false
5		false	$\neg$	
6		false		

$\neg$ example: 2

step	program	data	proc	proc-data
0	false $\neg$			
1	$\neg$	false		
2		false	$\neg$	
3			$\neg$	false
4			$\neg$	true
5		true	$\neg$	
6		true		

Puzzle 2.1

Write a single program that will test any number two times. First, test if it is greater than 5, then test if it equals 2. As an example if

3

 is the top card in your program it will go on the data stack first. Then your program should test if 3 is greater than 5, then test if 3 is equal to 2.

*Hint: You will need

dup

 and

swap

 in order to write a program that can test a number multiple times and rearrange the results on the **data** stack*

Chapter 3

Creating Our Own Functions

The art of programming is the art of organizing complexity.

– Edsger Dijkstra

What is a Function?

Imagine you are building a giant LEGO® creation. Would you plan out the entire LEGO® creation all at once? Or would you start with smaller components and combine them to make the final creation? Planning out the whole thing at once might work for some people, but once the creation gets large enough it becomes too complex for even the most intelligent people. We can hold only so much information in our brains at any given time. It is much easier to understand how to build and combine smaller components.

It is the same for programming. If we try to write out the entire program all at once, even the most intelligent people in the world will have trouble understanding it. We need a way to break the program into smaller components that we can combine - like LEGO® bricks. These smaller components are what we call functions.

We talked about some functions that are already part of the Programming Cards game. Those functions are good building blocks for small programs, but we want to write bigger, more interesting programs, so we need to be able to create our own functions.

You can think of a function as a small program. Remember that a program is just a list of instructions for a computer. So, a function is going to be a stack of cards. When we want to create a new function, all we have to do is take a stack of cards and assign a name to that stack. Take a blank card and write the name of your function on that card. Then, put the cards that make that function underneath or beside it. Let's work through an example together.

Let's start by putting our entire program onto the **program** stack, as we have been doing before, and then let's run the program.

Required cards

cards	count
--------------	--------------

+	1
*	1
100	1
5	2
dup	2
10	2

step	program	data	proc	proc-data
0	5 dup + dup *			
1	dup + dup *	5		
2	+ dup *	5	dup	
3	+ dup *		dup	5
4	+ dup *		dup	5 5
5	+ dup *	5	dup	5
6	+ dup *	5 5	dup	
7	+ dup *	5 5		
8	dup *	5 5	+	

9	dup *	5	+	5
10	dup *		+	5 5
11	dup *		+	10
12	dup *	10	+	
13	dup *	10		
14	*	10	dup	
15	*		dup	10
16	*		dup	10 10
17	*	10	dup	10
18	*	10 10	dup	
19	*	10 10		
20		10 10	*	
21		10	*	10
22			*	10 10
23			*	100
24		100	*	
25		100		

This program duplicates 5 and adds them together, resulting in 10. Then, it duplicates 10 and multiplies the two cards together, resulting in 100. Let's see how we can break this program up into our own functions.

Let's take the

dup

+

 and create a function out of it. Put the stack

dup

+

 somewhere where you have space. Now, let's give the function a name. You can name it whatever you like, but it is a good idea to always name functions in a way that helps you know what the function does. In this case, the function doubles whatever number is on the stack. So let's write the name double on the card.

Similarly, let's look at the

dup

*

 part of the program. Put

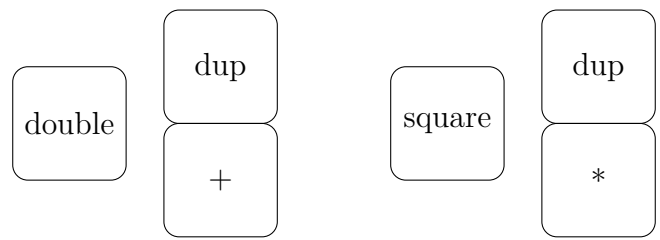
dup

*

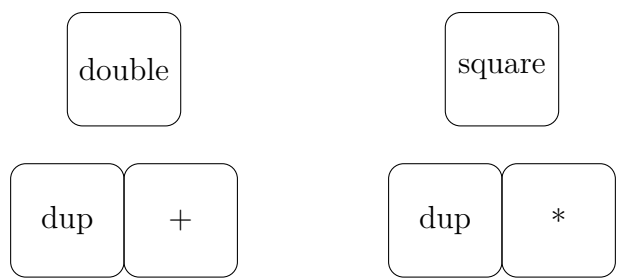
 in a stack somewhere to the side and write the name square on a card next to it. We have created two functions

to use as building blocks in our program.

Vertical Function Definition Layout



Horizontal Function Definition Layout



The diagrams above allow you to visualize how to layout your custom functions, but for the remainder of the book we will show custom functions using a table like the one below.

double	square				
<table><tr><td>dup</td></tr><tr><td>+</td></tr></table>	dup	+	<table><tr><td>dup</td></tr><tr><td>*</td></tr></table>	dup	*
dup					
+					
dup					
*					

Now, anywhere we see

dup
+

 in our program, we can replace it with the word double; and anywhere we see

dup
*

, we replace it with square.

old program	new program								
<table><tr><td>5</td></tr><tr><td>dup</td></tr><tr><td>+</td></tr><tr><td>dup</td></tr><tr><td>*</td></tr></table>	5	dup	+	dup	*	<table><tr><td>5</td></tr><tr><td>double</td></tr><tr><td>square</td></tr></table>	5	double	square
5									
dup									
+									
dup									
*									
5									
double									
square									

The old program has all the commands that the computer will execute, but it is kind of hard for us humans to follow. The new program is more like how we humans communicate. As you learn more about functions and how to build them, be on the lookout for how you can leverage custom functions to make your programs easier for humans to read.

Let’s run our new program. This time, we need to handle something new as the computer. As the computer, we need to know how to run a custom function when we see it. We execute

a custom function by copying the stack of cards from the function definition and putting them on top of the **program** stack.

double	square
dup +	dup *

Required cards

cards	count
double	1
square	1
100	1
*	1
+	1
5	2
10	2
dup	2

step	program	data
0	5 double square	
1	double square	5

Now, our custom function double is at the top of the **program** stack. So, to execute the double command, look at the double definition and copy that stack onto the **program** stack.

step	program	data	proc
2	dup + square	5	

Continuing...

step	program	data	proc	proc-data
3	+ square	5	dup	
4	+ square		dup	5
5	+ square		dup	5 5
6	+ square	5	dup	5
7	+ square	5 5	dup	
8	+ square	5 5		
9	square	5 5	+	
10	square	5	+	5
11	square		+	5 5
12	square		+	10
13	square	10	+	
14	square	10		

The custom function square is at the top of the **program** stack. We copy the cards from the square definition onto the **program** stack to execute the square command.

step	program	data	proc	proc-data
15	dup *	10		
16	*	10	dup	
17	*		dup	10
18	*		dup	10 10
19	*	10	dup	10

20	*	10 10	dup	
21	*	10 10		
22		10 10	*	
23		10	*	10
24			*	10 10
25			*	100
26		100	*	
27		100		

We didn't have to break up the program the way we did. We could have broken it up like this.

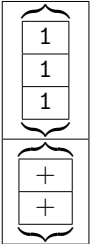
part1	part2	part3	program
5	dup + dup	*	part1 part2 part3

You can grab any part of your **program** stack, put it in a function definition, and have a building block. The program will always give the same result. Notice that this is a powerful tool for the programmer, but this creates slightly more work for the computer. The number of steps for the computer increases each time we define and use a custom function. It is up to the programmer to think about whether it is more important for them to make things easier on the computer or to make the program easier to understand. Most of the time, we want to make the program easier to understand.

Quoted Programs

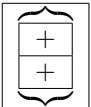
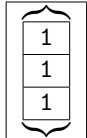
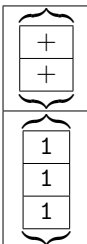
Quoting is another essential tool for writing programs. Quoting a program is similar to quoting an English sentence. In English we can quote something someone said by wrapping it in quotation marks. The sentence wrapped in quotation marks is separate from the containing sentence. We quote a program by wrapping it in a rubber band. Treat the quoted program like a single card. The command for the quoted program is just like numbers and strings; simply move the quoted program to the **data** stack.

quoted program example: 1

step	program	data
0		

Here, we have two quoted programs. The first one has the stack $\boxed{1} \boxed{1} \boxed{1}$, and the other has the stack $\boxed{+} \boxed{+}$. Whenever we have a quoted program, we will show it enclosed in . Every time you see a  just think that it is a rubber band. It is possible to have an empty  on the **data** stack. In that case there will be a rubber band without any cards in it on the **data** stack.

quoted program example: 1 continued...

step	program	data
1		
2		

Now, we can create functions that take functions as input and produce functions as output. With a custom function we use the name of the function to represent a pre-defined stack of cards. A quoted program just bundles the predefined stack of cards without giving it a name. In programming, we call that an anonymous function. Quoted programs are an extremely powerful tool. But this power is only complete with the ability to tell the computer to run the quoted program. There is a special command called unquote that does precisely that.

Unquote

`unquote` takes a quoted program (a stack inside a rubber band), removes the rubber band, places the stack all at once on the **program** stack, and discards the rubber band. There is no way to declare an exact output type. The output type depends on the quoted program.

Function	Input	Output
<code>unquote</code>	(Quoted Program)	(Output Type of Quoted Program)

Required cards

cards	count
<code>2</code>	1
<code>3</code>	1
<code>unquote</code>	2
<code>+</code>	2
<code>1</code>	3

unquote example: 1

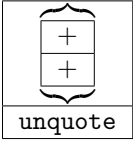
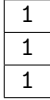
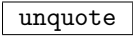
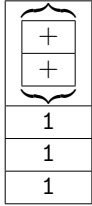
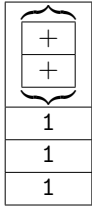
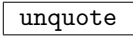
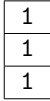
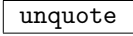
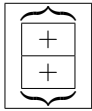
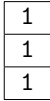
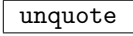
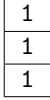
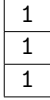
step	program	data	proc	proc-data
0	1 1 1 unquote + + unquote			
1	unquote + + unquote	1 1 1		

2	+ + unquote	1 1 1	unquote	
3	+ + unquote		unquote	1 1 1

When running a named function, we copy a stack of cards onto the **program** stack. When unquoting a quoted program, there is no copying. You simply remove the rubber band from the stack and place it directly on top of the **program** stack.

unquote example: 1 continued...

step	program	data	proc	proc-data
4	1 1 1 + + unquote		unquote	
5	1 1 1 + + unquote			
6	1 1 + + unquote	1		
7	1 + + unquote	1 1		

8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				

22			+	1 2
23			+	3
24		3	+	
25		3		

Designing Functions

Creating a function is easy. All we have to do is set aside a stack of cards and put a name on it. But remember our goal when creating functions. The goal is to create a program that is easier to understand. We want to break down our programs into smaller components like LEGO® bricks. It is easy to put a jumble of LEGO® bricks together, but it takes planning and careful consideration to make LEGO® components that fit together nicely. Let's go over a simple process for creating well-thought-out and connectable functions.

Steps of the Function Design Process

Designing a function follows three basic steps.

- 1st define the type
- 2nd think about specific inputs and outputs
- 3rd create the function stack

1. Define the Type

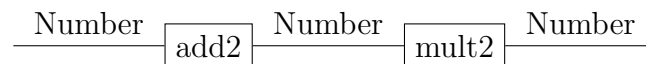
One of the helpful things about types is that they are guides for connecting functions. What does it mean to connect functions? It means we can run one function after another and the program still makes sense. Remember that when we run the process for a function, we take input from the **data** stack and put the result back on the **data** stack. Think about running two functions in a row. The result of the first function will be the input to the second function. Now, think about how we could create a nonsense program if we run a function that outputs a string and then try to run a function that takes numbers. The second function that wants a number will take the string and try to process it. The types of functions could have told us that we couldn't connect those two functions.

Consider the following functions.

Function	Input	Output
<code>add2</code>	(Number)	(Number)
<code>mult2</code>	(Number)	(Number)

`add2` is a function that takes a number and adds 2 to it, then outputs the resulting number. `mult2` is a function that takes a number, multiplies it by 2, and outputs the resulting number. We know that these two functions can connect because the output of one matches the input of the other.

Here is a diagram to help you visualize how functions connect based on their types.



Here is an example of two functions that can not be connected. Without even knowing what these functions do, can you explain why we can't connect them?

Function	Input	Output
<code>reverse</code>	(String)	(String)
<code>negate</code>	(Number)	(Number)



2. Think About Specific Inputs and Outputs

After thinking about the type of a function, which is a general way of thinking about inputs and outputs, we think about *specific* inputs and outputs. That is to say that we think about what-if scenarios. What should the output be if we give it an input of 1? What should the output be if we give it an input of 2? We should create a table of specific inputs and outputs like the one below.

Input	Output
0	0
1	2
2	4
3	6
100	200
150	300

The table above shows a handful of specific examples. It gives the input and the expected output for the `mult2` function. However, it doesn't show all possible inputs. Showing all possible inputs would take an infinite amount of time because there is an infinite number of inputs. So, we must use our judgment as programmers to determine the most valuable examples and how many we need.

It's your turn to create a table for the `add2` function.

Input	Output
?	?
?	?
?	?

3. Create the Function Stack

The type and the examples table are like the blueprints for a function. Once we have the blueprints figured out, then we can start building. We build a set of instructions that will respect the input and output types we established. We also build those instructions so that the behavior of the function matches the input and output examples we listed in our table. Let's create the `add2` and `mult2` functions.

<code>add2</code>	<code>mult2</code>
<code>2</code> <code>+</code>	<code>2</code> <code>*</code>

Puzzle 3.1

Follow the function design process to create a function that takes a number and adds 10 to it.

Puzzle 3.2

Follow the function design process to create a function that takes a number as input and outputs the cube of that number. The function should work with any number as input. For example `2` `cube` or `3` `cube`.

Hint: Refer to the double function from earlier in this chapter, and review the examples in Chapter 1 where `+` comes at the end.

The Master Function Builder

In the LEGO® universe, they use the term master builder to refer to someone who is an elite LEGO® creator. A master builder is creative, resourceful, and can turn their imagination into reality. In the programming world, the function is our LEGO® brick; therefore, to be a master programmer, you need to be a master function builder. As programmers, it can sometimes be challenging to turn our imagination into reality. Sometimes, we get stuck and need help figuring out where to start. You can always start with a function. Mastering the function design process is how you become a master function builder.

Chapter 4

Conditionals

Sometimes it is the people no one imagines anything of who do the things that no one can imagine.

– Alan Turing

Why Conditionals are Important

Envision yourself in a distant past as either a king or a queen. Your kingdom is unlike any other, where every person in this kingdom has willingly pledged to obey every command with exactness. Everyone waits for you to decide on how to do anything. The farmers look to you to know what animals to raise and what crops to grow. The merchants wait for you to tell them what to sell and at what price.

Let's pretend that you command everyone to honor your birthday as a special holiday, and on this holiday, everyone must have a raspberry jelly-filled donut for dessert. For most people, this is a wonderful command because most people love raspberry jelly donuts. However, some people in your kingdom are allergic to raspberries, so they will become extremely ill if you make them eat raspberries. Remember that if you command it, they will do it, even if it makes them sick. But you are a kind and benevolent monarch so you modify your command to be a conditional command. Your new command is if you are *not* allergic to raspberries, then you must eat a raspberry jelly-filled donut for dessert, but if you *are* allergic, then you must eat a chocolate cream pie.

As you might imagine, your kingdom would have many problems if you did not know how to give a conditional command. Your kingdom will only be as good as your ability to think ahead and construct conditional commands. What should your loyal subjects do if there is a flood? What should they do if there is a drought? In your kingdom you have the power to command anything you wish and the responsibility to make fully thought-out commands.

You can imagine how nice it would be to live in a kingdom like this - where everyone does exactly what you say. In reality, though, we can't control people. Looking back at even the most powerful kings and queens, they could never control everyone to make them do exactly what they wanted. In real life you can't control people nor should you try. However, you have an army of servants that will obey your every command when you are a programmer.

These servants are computers. If you think about it, you have more power than the kings and queens of olden times. The only limit is your ability to craft instructions for a computer to follow.

Grammar of Conditionals

To be good at crafting English sentences we study the parts that make up a sentence. We study nouns, verbs, and the patterns and structure of a sentence. To be good at crafting conditional commands, we must examine the patterns, structure, and rules of a conditional command. Just as English has a grammar, a conditional command has a grammar. The grammar we use to program machines is much smaller and simpler than the grammar we use to construct sentences. Let's break down the conditional grammar.

The Structure of a Conditional

IF *{condition program}* THEN *{true program}* ELSE *{false program}*

We have the words IF THEN and ELSE in all capital letters. This is to demonstrate that these words are special. We use them to know that we are looking at a conditional. The *{condition program}* *{true program}* and *{false program}* are in curly braces to demonstrate that we can fill it in with our own programs. That is the key here. A conditional works with three programs and each program has a special purpose.

1. *{condition program}*:

A condition is any program that, when run, results in a boolean. This program is run first to decide which of the other two programs to run. We use a boolean to signal to the computer which one to run because a boolean only has two options. If the condition program results in `true` then we run the *{true program}* and if it results in `false` then we run the *{false program}*.

2. *{true program}*:

The true program is only run when the condition program outputs true.

3. *{false program}*:

The false program is only run when the condition program outputs false.

Let's look at how a machine would think through the following conditional.

IF *is tired* THEN *nap* ELSE *play*

The machine would first run the program *is tired* and check if it is true or false. If it is true, then the machine knows that it should run the *nap* program. If it is false, then it should run the *play* program.

Conditionals in the Programming Cards Game

It's time for the computer to learn about two new cards, `ifte` and `etfi`. Up until now most the processes take cards from the **data** stack, process the result and then put the result back on the **data** stack. `unquote` was an exception to this rule, where it placed the result onto the **program** stack instead of the **data** stack. `ifte` and `etfi` are two more exceptions. They take data from the **data** stack but they interact with more stacks as well. The process is still simple, but it may take a few practice runs to get the hang of it.

The `ifte` and `etfi` cards are a pair and always go together. If you do not put them together then the program will be nonsense. Ifte is short for IF THEN ELSE. Etfi is the word ifte spelled backwards. The computer uses `ifte` to begin the conditional and `etfi` to end the conditional.

The way to construct a conditional program is to take the grammar for a conditional and morph it into reverse polish notation like this.

$\underbrace{\text{false program}} \underbrace{\text{true program}} \underbrace{\text{condition program}} \text{ifte} \text{etfi}$

That is the general pattern of creating a conditional program. Or we could say that this is the grammar. The `ifte` `etfi` pair at the right is like the IF THEN ELSE smashed together. The $\underbrace{\text{condition program}}$ $\underbrace{\text{true program}}$ and $\underbrace{\text{false program}}$ follow the `ifte` from right to left. Now, we can use that pattern to create the program we want to run. Take this general pattern and fill it in with our own three programs.

$\underbrace{\text{"play"}} \underbrace{\text{"nap"}} \underbrace{\text{isTired?}} \text{ifte} \text{etfi}$

In reverse polish notation the computer reads the commands from left to right, but it is easier for programmers to read from right to left. Starting from the right we see the pair of `ifte` `etfi` cards and we as programmers can read that aloud as "IF" or "begin conditional". Then continue reading right to left and it will sound more like a natural conditional sentence. I would read the program above from right to left like this: If is tired then nap else play.

Here is an example program to practice with. We are going to simplify our condition program and pretend that we already know if we are tired or not.

program	
	"play" "nap" true ifte etfi

IFTE

IFTE takes three quoted programs as input. It moves the top two quoted programs (one at a time) from the **proc-data** stack to the **ifte-data** stack. Then it takes the third and final quoted program on the **proc-data** stack and unquotes it. Remember that unquote means to run the program inside the rubber bands. So remove the rubber band and place it's contents on the **program** stack.

Function	Input	Output
ifte	ELSE THEN IF	()

ETFI

ETFI takes one boolean card from the **data** stack. If it is

true

 then it unquotes the top quoted program from the **ifte-data** stack and discards the second quoted program. If it is

false

 then it unquotes the second quoted program from the **ifte-data** stack and discards the top quoted program.

Function	Input	Output
ifte	Boolean	(THEN OR ELSE)

Let's run through the practice example.

Required cards

cards	count
"play"	1
ifte	1
true	1
etfi	1
"nap"	1

ifte example: 1

step	program	data	proc
------	---------	------	------

0	"play" "nap" true ifte etfi		
1	"nap" true ifte etfi	"play"	
2	true ifte etfi	"nap" "play"	
3	ifte etfi	true "nap" "play"	

The next command is `ifte` so start the process by moving `ifte` to the **proc** stack. Then move the three quoted programs from the **data** stack to the **proc-data** stack one at a time. We could summarize this action and say the `ifte` process *takes* three quoted programs.

ifte example: 1

step	program	data	proc	proc-data
4	etfi	true "nap" "play"	ifte	

5	etfi	"nap" "play"	ifte	true
6	etfi	"play"	ifte	"nap" true
7	etfi		ifte	"play" "nap" true

Run the ifte process by moving the top two quoted programs to the **ifte-data** stack (one at a time), and then unquote the third quoted program. This completes the ifte process so follow it up by clearing ifte from the **proc** stack.

ifte example: 1

step	program	data	proc	proc-data	ifte-data
9	etfi		ifte	"nap" true	"play"
10	etfi		ifte	true	"nap" "play"
11	true etfi		ifte		"nap" "play"
12	true etfi				"nap" "play"

That is it for `ifte`. Now run the next command from the **program** stack.

ifte example: 1 continued...

step	program	data	proc	proc-data	ifte-data
13	<code>etfi</code>	<code>true</code>			"nap" "play"

The condition program was a single `true` card, and as we can see, the result of the condition program ends up on the **data** stack waiting for the `etfi` card to read it. The `etfi` card uses the result of the condition program to decide which of the two quoted programs on the **ifte-data** stack to run. In this case it will run the *{true program}*, which will always be the top quoted program on the **ifte-data** stack. Then it will discard the *{false program}* which will always be the second quoted program on the **ifte-data** stack.

ifte example: 1 continued...

step	program	data	proc	proc-data	ifte-data
14		<code>true</code>	<code>etfi</code>		"nap" "play"
15			<code>etfi</code>	<code>true</code>	"nap" "play"
16	<code>"nap"</code>		<code>etfi</code>		
17	<code>"nap"</code>				
18		<code>"nap"</code>			

The final output is `"nap"`. Run it again with the condition program of `false`.

Take a minute to discuss the process of `ifte` and `etfi` before moving on.

Question: What should the output of the condition program be?

Answer: The condition program must always output a single boolean

Question: Why do the three inputs to `ifte` need to be quoted programs?

Answer: They need to be quoted programs so that the computer can treat them like data. If a card isn't quoted it is processed as it is read from the **program** stack. So quoting

a card, or stack of cards is like telling the computer to ignore processing it for now, and we may decide to process it later.

Question: What is the purpose of having an **ifte-data** stack.

Answer: The stacks are the computer’s memory. The computer uses the **ifte-data** stack to remember which program to run depending on the condition. Theoretically we could teach the computer to use only one **data** stack. However, that would make the programs much more complicated, and require many more steps to process conditionals. We use a sparate special data stack that is only ever used by the ifte process to make things simpler.

Ifte With a Bigger *condition program*

We were just short circuiting the *istired?* program. Instead of a full program we just used a single **true** or **false**. Substituting a smaller simpler program for part of the overall program is a useful trick to understand how the program will behave. Now let’s work on a conditional that does more work before deciding which program to run.

Required cards

cards	count
1	1
ifte	1
5	1
6	1
≡	1
false	1
etfi	1
2	1

ifte example: 2

step	program	data	proc	proc-data	ifte-data
------	---------	------	------	-----------	-----------

0	6 2 1 5 ≡ ifte etfi				
1	2 1 5 ≡ ifte etfi	6			
2	1 5 ≡ ifte etfi	2 6			
3	5 ≡ ifte etfi	1 2 6			
4	ifte etfi	5 ≡ 1 2 6			

5	etfi	5 ≡ 1 2 6	ifte		
6	etfi	1 2 6	ifte	5 ≡	
7	etfi	2 6	ifte	1 5 ≡	
8	etfi	6	ifte	2 1 5 ≡	
9	etfi	6	ifte	2 1 5 ≡	
10	etfi	6	ifte	1 5 ≡	2
11	etfi	6	ifte	5 ≡	1 2

12	5 ≡ etfi	6	ifte		1 2
13	5 ≡ etfi	6			1 2
14	≡ etfi	5 6			1 2
15	etfi	5 6	≡		1 2
16	etfi	6	≡	5	1 2
17	etfi		≡	6 5	1 2
18	etfi		≡	false	1 2
19	etfi	false	≡		1 2
20	etfi	false			1 2
21		false	etfi		1 2

22			etfi	false	1 2
23	2		etfi		
24	2				
25		2			

As the programmer it is important to think about what we are leaving behind on the **data** stack *after* running a command and what is available on the **data** stack *before* running a command. Just as we think about how functions are connectable we should think about how conditionals are connectable - or any part of our program for that matter. We need to think about the input type for the *conditional program* program as well as the input type for the *true program* and *false program*.

One More Time

One more example to make sure we understand the steps for this command, as well as how to think about this as programmers.

program	
	2
	2
	+
	"fish"
	"four"
	4
	≡
	ifte
	etfi

Interestingly the computer can handle larger programs easily. Processing takes a bit more time, but the steps are simple. The computer executes the program one command at a time. It doesn't need to peek ahead; it uses stacks to remember everything it needs. Contrast this with the programmer. As the programmer, it is easy to understand small programs and hard to understand large programs. The programmer must think about how each function connects with the next across the entire program. Now is an excellent time to practice some techniques to understand longer programs.

In the previous chapter we learned how to break up larger programs into smaller functions. We can replace part of a program with one word. Choosing words wisely can make our program read more like a story. Of course, in the Programming Cards game, it is a story that we read backward.

If we know that we can break up the parts of the program and make a function out of them, then we can also think about parts of the program in isolation. Once we understand the smaller parts, it is easier to think about the program as a whole.

What if we split up the program like this?

first part	second part
2 2 +	"fish" "four" 4 ≡ ifte etfi

If we look at the whole program all at once, all the commands bleed into each other. Now, we can see that there is a logical separation between the ideas in this program. The first idea is that 2 and 2 will be added together. We all know that will result in 4, but we could just say that whatever the result of that first part is, it will end up on the data stack.

The second part is a conditional. In this conditional, we can see that the idea is to test whether whatever is on the data stack is equal to 4. If it is, we will run the "four" program, and if it isn't, we will run the "fish" program.

Predict what you think the result will be, and then run the program.

Required cards

cards	count
+	1
"fish"	1
ifte	1
≡	1
true	1
etfi	1
"four"	1
2	2

4	2
---	---

ifte example: 3

step	program	data	proc	proc-data	ifte-data
0	2 2 + "fish" "four" 4 ≡ ifte etfi				
1	2 + "fish" "four" 4 ≡ ifte etfi	2			
2	+ "fish" "four" 4 ≡ ifte etfi	2 2			

3	"fish" "four" 4 ≡ ifte etfi	2 2	+		
4	"fish" "four" 4 ≡ ifte etfi	2	+	2	
5	"fish" "four" 4 ≡ ifte etfi		+	2 2	
6	"fish" "four" 4 ≡ ifte etfi		+	4	

7	"fish" "four" 4 ≡ ifte etfi	4	+		
8	"fish" "four" 4 ≡ ifte etfi	4			
9	"four" 4 ≡ ifte etfi	"fish" 4			
10	4 ≡ ifte etfi	"four" "fish" 4			
11	ifte etfi	4 ≡ "four" "fish" 4			

12	etfi	4 "four" "fish" 4	ifte		
13	etfi	"four" "fish" 4	ifte	4 ≡	
14	etfi	"fish" 4	ifte	"four" 4 ≡	
15	etfi	4	ifte	"fish" "four" 4 ≡	
16	etfi	4	ifte	"fish" "four" 4 ≡	
17	etfi	4	ifte	"four" 4 ≡	"fish"
18	etfi	4	ifte	4 ≡	"four" "fish"

19	4 ≡ etfi	4	ifte		"four" "fish"
20	4 ≡ etfi	4			"four" "fish"
21	≡ etfi	4 4			"four" "fish"
22	etfi	4 4	≡		"four" "fish"
23	etfi	4	≡	4	"four" "fish"
24	etfi		≡	4 4	"four" "fish"
25	etfi		≡	true	"four" "fish"
26	etfi	true	≡		"four" "fish"
27	etfi	true			"four" "fish"
28		true	etfi		"four" "fish"

29			etfi	true	"four" "fish"
30	"four"		etfi		
31	"four"				
32		"four"			

Yes, two plus two is four. If your computer gives a different answer you may need to take it to the repair shop.

Puzzle 4.1

Create a function that takes a number as input. If the number is between 2 and 4 then output the string "inside". If the number is not between 2 and 4 then output the string "outside".

Hint: Work on creating a function that can test if a number is between 2 and 4. Then use that function as your condition program.

Chapter 5

Understanding Recursion

Nothing new had been done in Logic since Aristotle!

– Kurt Gödel

The Foundation of Computing

About 100 years ago, before modern computers existed, some mathematicians were trying to find a way to build machines that could solve their math problems. Their math problems were different from yours and mine. The problems they were looking to solve were deep, complicated questions about logic and the universe. Their question was, "Can we build a machine that can solve all our problems automatically?". If they could build this machine, they wouldn't have to do all the work themselves and they could use the machine to learn the mysteries of the universe. There were many mathematicians, but three of the most famous were Kurt Gödel, Alonzo Church, and Alan Turing.

These three mathematicians set out independently to solve the problem of building the theoretical machine. Using logic, they discovered that creating a mechanical device that could automate some but not all of their work was indeed possible. These mathematicians are known as the founders of modern computing. Our computers are based on their work.

The first step of their work was to define an algorithm. Today, we use algorithm as a fancy word for program, but back then, when there were no computers or programmers, they had to figure out what it meant to write instructions for a mechanical device to execute. A critical part of what they discovered was that in order to program a computer, two things were necessary: conditionals and recursion.

We talked about conditionals before. A conditional gives us the ability to make choices in the program. In the Programming Cards game, we use `ifte` `etfi` for making choices. What is recursion, though?

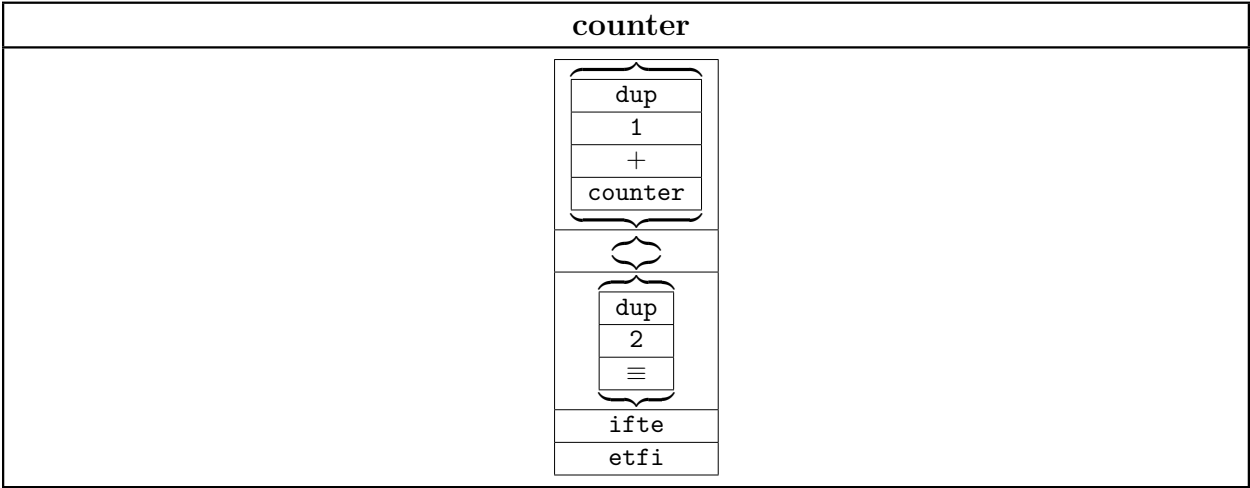
Recursion is simply when a function references itself.

Counting

Let’s create a function called counter. In this function, we are going to count to 2. The final result on the **data** stack should be

2
1
0

.



Required cards

cards	count
false	1
true	1
etfi	1
ifte	1
≡	1
counter	1
+	1
0	2
dup	2
2	3
1	3

step	program	data		
0	<table><tr><td>0</td></tr><tr><td>counter</td></tr></table>	0	counter	
0				
counter				

1	counter	0
---	---------	---

Like before, we replace the counter card on the **program** stack with a copy of the counter definition.

step	program	data	proc	proc-data	ifte-data
2	dup 1 + counter dup 2 ≡ ifte etfi	0			
3	dup 2 ≡ ifte etfi	dup 1 + counter 0			
4	dup 2 ≡ ifte etfi	dup 1 + counter 0			
5	ifte etfi	dup 2 ≡ dup 1 + counter 0			

6	etfi	dup 2 ≡ dup 1 + counter 0	ifte		
7	etfi	dup 1 + counter 0	ifte	dup 2 ≡	
8	etfi	dup 1 + counter 0	ifte	dup 2 ≡	
9	etfi	0	ifte	dup 1 + counter dup 2 ≡	
10	etfi	0	ifte	dup 1 + counter dup 2 ≡	

11	etfi	0	ifte	dup 2 ≡	dup 1 + counter
12	etfi	0	ifte	dup 2 ≡	dup 1 + counter
13	dup 2 ≡ etfi	0	ifte		dup 1 + counter
14	dup 2 ≡ etfi	0			dup 1 + counter
15	2 ≡ etfi	0	dup		dup 1 + counter
16	2 ≡ etfi		dup	0	dup 1 + counter
17	2 ≡ etfi		dup	0 0	dup 1 + counter

18	2 ≡ etfi	0	dup	0	dup 1 + counter
19	2 ≡ etfi	0 0	dup		dup 1 + counter
20	2 ≡ etfi	0 0			dup 1 + counter
21	≡ etfi	2 0 0			dup 1 + counter
22	etfi	2 0 0	≡		dup 1 + counter
23	etfi	0 0	≡	2	dup 1 + counter
24	etfi	0	≡	0 2	dup 1 + counter

25	etfi	0	≡	false	dup 1 + counter
26	etfi	false 0	≡		dup 1 + counter
27	etfi	false 0			dup 1 + counter
28		false 0	etfi		dup 1 + counter
29		0	etfi	false	dup 1 + counter
30	dup 1 + counter	0	etfi		
31	dup 1 + counter	0			
32	1 + counter	0	dup		

33	1 + counter		dup	0	
34	1 + counter		dup	0 0	
35	1 + counter	0	dup	0	
36	1 + counter	0 0	dup		
37	1 + counter	0 0			
38	+ counter	1 0 0			
39	counter	1 0 0	+		
40	counter	0 0	+	1	
41	counter	0	+	0 1	
42	counter	0	+	1	
43	counter	1 0	+		
44	counter	1 0			

Now the **program** stack looks like it did before we ran the counter function; except this time there is an extra card on the **data** stack. Each time the counter function runs, it first checks if the card on top of the **data** stack is

2

. If it is

2

 then it will stop running itself. If it isn't

2

 then it will count up, and run itself again.

step	program	data	proc	proc-data	ifte-data
------	---------	------	------	-----------	-----------

45	dup 1 + counter dup 2 ≡ ifte etfi	1 0			
46	dup 2 ≡ ifte etfi	dup 1 + counter 1 0			
47	dup 2 ≡ ifte etfi	dup 1 + counter 1 0			
48	ifte etfi	dup 2 ≡ dup 1 + counter 1 0			

49	etfi	dup 2 ≡ dup 1 + counter 1 0	ifte		
50	etfi	dup 1 + counter 1 0	ifte	dup 2 ≡	
51	etfi	dup 1 + counter 1 0	ifte	dup 2 ≡	
52	etfi	1 0	ifte	dup 1 + counter dup 2 ≡	

53	etfi	1 0	ifte	dup 1 + counter dup 2 ≡	
54	etfi	1 0	ifte	dup 2 ≡	dup 1 + counter
55	etfi	1 0	ifte	dup 2 ≡	dup 1 + counter
56	dup 2 ≡ etfi	1 0	ifte		dup 1 + counter
57	dup 2 ≡ etfi	1 0			dup 1 + counter
58	2 ≡ etfi	1 0	dup		dup 1 + counter

59	2 ≡ etfi	0	dup	1	dup 1 + counter
60	2 ≡ etfi	0	dup	1 1	dup 1 + counter
61	2 ≡ etfi	1 0	dup	1	dup 1 + counter
62	2 ≡ etfi	1 1 0	dup		dup 1 + counter
63	2 ≡ etfi	1 1 0			dup 1 + counter
64	≡ etfi	2 1 1 0			dup 1 + counter
65	etfi	2 1 1 0	≡		dup 1 + counter

66	etfi	1 1 0	≡	2	dup 1 + counter
67	etfi	1 0	≡	1 2	dup 1 + counter
68	etfi	1 0	≡	false	dup 1 + counter
69	etfi	false 1 0	≡		dup 1 + counter
70	etfi	false 1 0			dup 1 + counter
71		false 1 0	etfi		dup 1 + counter
72		1 0	etfi	false	dup 1 + counter

73	dup 1 + counter	1 0	etfi		
74	dup 1 + counter	1 0			
75	1 + counter	1 0	dup		
76	1 + counter	0	dup	1	
77	1 + counter	0	dup	1 1	
78	1 + counter	1 0	dup	1	
79	1 + counter	1 1 0	dup		
80	1 + counter	1 1 0			
81	+ counter	1 1 1 0			
82	counter	1 1 1 0	+		
83	counter	1 1 0	+	1	
84	counter	1 0	+	1 1	
85	counter	1 0	+	2	

86	counter	2 1 0	+		
87	counter	2 1 0			

Here it is again. The counter program put itself back on the **program** stack as the next command for the computer to run. This time the **data** stack has a 2 on top. So this is the last time the counter program will run itself. This time it will choose to run the *{true program}* which is to do nothing.

step	program	data	proc	proc-data	ifte-data
88	dup 1 + counter dup 2 ≡ ifte etfi	2 1 0			
89	dup 1 + counter dup 2 ≡ ifte etfi	dup 1 + counter 2 1 0			
90	dup 1 + counter dup 2 ≡ ifte etfi	dup 1 + counter dup 2 ≡ ifte etfi 2 1 0			

91	ifte etfi	dup 2 ≡ dup 1 + counter 2 1 0			
92	etfi	dup 2 ≡ dup 1 + counter 2 1 0	ifte		
93	etfi	dup 1 + counter 2 1 0	ifte	dup 2 ≡	
94	etfi	dup 1 + counter 2 1 0	ifte	dup 2 ≡	

95	etfi	2 1 0	ifte	dup 1 + counter dup 2 ≡	
96	etfi	2 1 0	ifte	dup 1 + counter dup 2 ≡	
97	etfi	2 1 0	ifte	dup 2 ≡	dup 1 + counter
98	etfi	2 1 0	ifte	dup 2 ≡	dup 1 + counter
99	dup 2 ≡ etfi	2 1 0	ifte		dup 1 + counter
100	dup 2 ≡ etfi	2 1 0			dup 1 + counter

101	2 ≡ etfi	2 1 0	dup		dup 1 + counter
102	2 ≡ etfi	1 0	dup	2	dup 1 + counter
103	2 ≡ etfi	1 0	dup	2 2	dup 1 + counter
104	2 ≡ etfi	2 1 0	dup	2	dup 1 + counter
105	2 ≡ etfi	2 2 1 0	dup		dup 1 + counter
106	2 ≡ etfi	2 2 1 0			dup 1 + counter
107	≡ etfi	2 2 2 1 0			dup 1 + counter

108	etfi	2 2 2 1 0	≡		dup 1 + counter
109	etfi	2 2 1 0	≡	2	dup 1 + counter
110	etfi	2 1 0	≡	2 2	dup 1 + counter
111	etfi	2 1 0	≡	true	dup 1 + counter
112	etfi	true 2 1 0	≡		dup 1 + counter
113	etfi	true 2 1 0			dup 1 + counter
114		true 2 1 0	etfi		dup 1 + counter

115		2 1 0	etfi	true	dup 1 + counter
116		2 1 0	etfi		
117		2 1 0			

Finally! The stack has `[2 1 0]`, just like we wanted. Take a minute to talk or think about why the counter function behaves the way that it does.

Recursion is simple, but the uses of recursion are profound and complex. This book can only cover a small portion of what there is to know about recursion, but the more you know, the more powerful a programmer you will be. If you are serious about your programming education, then this is one of the topics you should spend a lot of time learning.

Remember that I said to be a master programmer you need to be a master function builder? This includes knowing how to be a master *recursive* function builder too. Many programmers learn the most basic form of recursion and then stop learning about it. Many programmers don't bother to learn recursion because the programming language they learned uses a particular form of repeating commands called a loop. Recursion is different than a loop and is much more powerful if the programmer knows how to use it. If you don't understand recursion, your ability to stack smaller programs on top of each other is limited to the non-repeating parts of your program. Whereas, if you know recursion and how to combine recursive functions, your program can be well organized all the way through. Even the parts of the program that repeat can be connectable components. The following section is a primer on learning to be a master recursive function builder.

The Master Recursive Function Builder

There is a fun math operator called factorial. The factorial operator is the `!` symbol. For example, `3!` means three factorial. We perform the factorial operation by multiplying all the numbers from 1 to the number. So `3!` is `1 * 2 * 3`, and `5!` is `1 * 2 * 3 * 4 * 5`. Let's write a factorial function. We want a function that will output the factorial of any number. This section helps you learn how to create a recursive function by following the function design process from Chapter 3.

Step 1: Define the Factorial Type

Our factorial function will take a number as input and output a single number.

Function	Input	Output
<code>fact</code>	(Number)	(Number)

Step 2: Specific Factorial Examples

The table below has the examples of inputs and outputs. Fill in the question marks on your own.

Inputs	outputs
1	1
2	2
3	6
4	?
5	?
6	?

Step 3: Create the Function Stack

Alright, you know the drill. Create the function stack for factorial. It's okay; I'll wait... What do you mean you don't know how? I've shown you everything you need to know to create this function! Remember how we had a whole section on designing functions?

It's okay if you don't know what to do here. This is the life of the programmer. You will run into situations where you are not sure how to continue. In these situations the programmer needs to know how to think through the problem. When you are first learning programming it is hard to see the patterns for solving problems. As you are more practiced in programming you will have the skills and confidence to solve any problem. Recursive functions have a pattern and you just need to find it.

There are two parts to writing recursive functions. The first part is the recursive case, and the second is the base case. The recursive case is when the function calls itself again. In other words, the recursive case is when we want to *repeat* something. The base case is when we want to *stop* repeating. Conditionals and recursion go hand in hand. We use the conditional to decide when to repeat and when to stop repeating.

Factorial Recursive Case

There is a repetitive action in calculating a factorial. Do you know what it is? $4!$ is $1 * 2 * 3 * 4$. First, we multiply $1 * 2$. Then, we multiply $2 * 3$. Then, we multiply $6 * 4$. What are we repeating? We are repeating the multiplication. But we aren't repeating multiplying the same number over and over. We repeat multiplying by supplying two new numbers to the multiplication function each time. One of the numbers is the result of the previous multiplication step and the other number is increasing by one each time.

With the Programming Cards game we can use the cards to help us think about the factorial process. Run each of these programs to see if you can find the repeating patterns.

fact1	fact2	fact3	fact4
1	1 2 *	1 2 3 * *	1 2 3 4 * * *

Notice how in the fact4 function we repeat the `*` command three times? This is a good clue that it needs to be in our recursive case. However, it's not enough information. Another pattern we see is that one number that goes into the multiplication step changes by 1 each time. It still isn't capturing the full pattern though. Yet another pattern we see is that all the functions do the same work at the beginning. Fact2 is like fact1 with one more step. fact3 is fact2 with one more step, and fact4 is fact3 with one more step. This is the pattern we are looking for.

Let's start with the last pattern. Whenever we have a function that does the same work we can substitute the name of that function for that work.

fact1	fact2	fact3	fact4
1	2 fact1 *	3 fact2 *	4 fact3 *

We know that fact4 is the same as $4 * \text{fact3}$ and that fact3 is the same as saying $3 * \text{fact2}$. So, the factorial of a number is the same as the number multiplied by the factorial of the preceding number. That is how we say it in English, but how do we turn that into a program? Run the following program and see if you can catch what is happening.

fact
dup 1 — fact *

Let's test it out.

Required cards

cards	count
fact	1
—	1

dup	1
3	2
2	2
1	2
*	3

step	program	data	proc	proc-data
0	3 fact			
1	fact	3		
2	dup 1 — fact *	3		
3	1 — fact *	3	dup	
4	1 — fact *		dup	3
5	1 — fact *		dup	3 3
6	1 — fact *	3	dup	3
7	1 — fact *	3 3	dup	

8	1 — fact *	3 3		
9	— fact *	1 3 3		
10	fact *	1 3 3	—	
11	fact *	3 3	—	1
12	fact *	3	—	3 1
13	fact *	3	—	2
14	fact *	2 3	—	
15	fact *	2 3		
16	dup 1 — fact * *	2 3		
17	1 — fact * *	2 3	dup	
18	1 — fact * *	3	dup	2
19	1 — fact * *	3	dup	2 2

20	1 — fact * *	2 3	dup	2
21	1 — fact * *	2 2 3	dup	
22	1 — fact * *	2 2 3		
23	— fact * *	1 2 2 3		
24	fact * *	1 2 2 3	—	
25	fact * *	2 2 3	—	1
26	fact * *	2 3	—	2 1
27	fact * *	2 3	—	1
28	fact * *	1 2 3	—	
29	fact * *	1 2 3		

Ok, stop there. As a side note this program runs forever. You might want to spend some time to think about why that is the case. For now though, there are a couple of important things to notice about our fact definition. First, we use `dup` on whatever number was already on the **data** stack and then subtract 1 from it. The result of doing this recursively

creates an ever decreasing stack of numbers.

The second thing to notice is that the `fact` function definition keeps adding the `*` card on the **program** stack, but we never get to that command because we have to run `fact` first. The problem is that `fact` doesn't know when to stop repeating.

Factorial Base Case

Every time we enter a recursive step we need to check whether we should stop or continue. Recall how the stacks looked at the end of the recursive case.

program	data
fact	1
*	2
*	3

We see the `fact` card on top of the **program** stack. Imagine all the cards stay the same except the `fact` card. What could we replace `fact` with and get the correct result? A simple answer is to replace `fact` with nothing. That can be our base case: put no cards on the stack. That would work because that would allow the continuation of the program to multiply all the numbers on the **data** stack together.

The question now is when to do the base case and when to recurse. A good place to stop is when we see 1 on top of the **data** stack.

fact
dup 1 — fact * dup 1 ≡ ifte etfi

Before we move on, can you point out the base case and recursive case in this function? The base case is the *{true program}*. The recursive case is the *{false program}*. We decide which one to do based on the result of the *{condition program}*.

I would congratulate you but we aren't done yet. You see, our factorial function needs to be corrected. It is mostly correct, but mostly correct is still wrong. Factorial is actually defined for all positive numbers and 0. Our factorial function doesn't work with 0. What is 0! though? $0! = 1$. That is the real base case, so we need to modify our program to stop

when it sees 0 top of the **data** stack. Let's ammend our examples table to work with the number zero as input.

Inputs	outputs
0	1
1	1
2	2
3	6

Imagine for a minute if we didn't have $0! = 1$ as the base case, the following would happen, where 0 messes up the multiplication chain we recursively built up.

step	program	data
0	fact * * *	0 1 2 3
1	* * *	0 1 2 3
2	* *	0 2 3
3	*	0 3
4		0

So let's try again. What can replace fact but will keep the rest of our multiplication steps? Let's start by defining the correct fact0. To find the correct fact0 let's play along with the cards.

step	program	data
0	0 fact0	
1	fact0	0

The fact0 function should output 1. So it needs to consume the 0, and put a 1 on the **data** stack. Let's replace our `fact0` card with a program that will replace the `0` with a `1`.

step	program	data
0	pop 1	0
1	1	
2		1

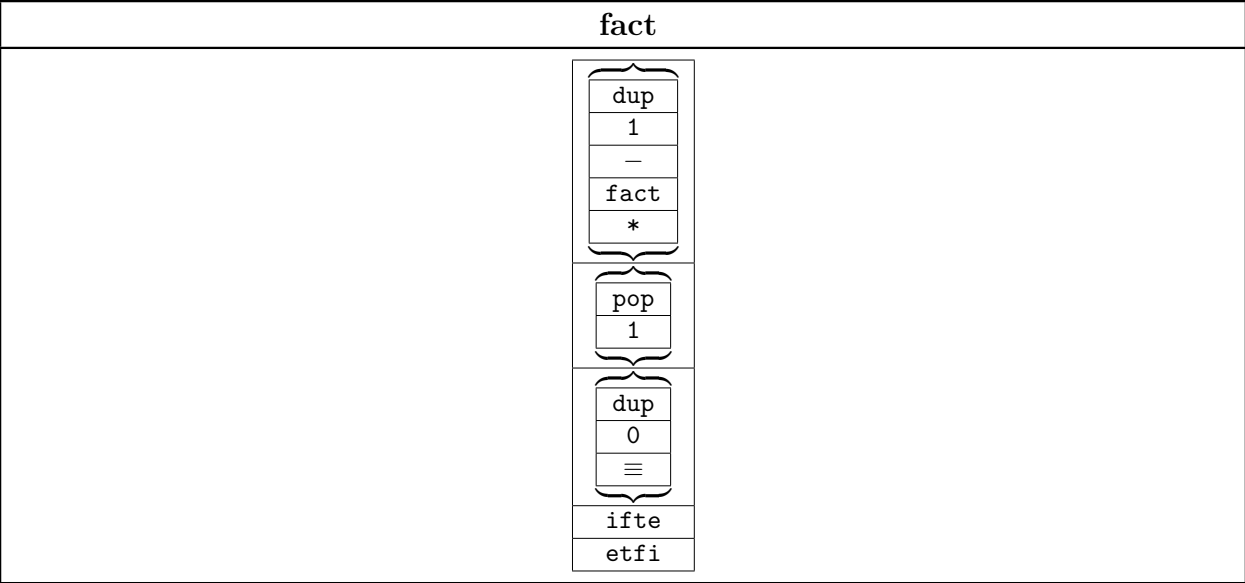
That is our real base case. Now, modify our fact definition to this.

fact
dup 1 — fact * pop 1 dup 0 ≡ ifte etfi

Congratulations! If we see 0 on top of the stack, we stop recursing, and we put 1 on the stack. $0! = 1$ after all, it makes sense because multiplying by 1 doesn't mess up the rest of the multiplication steps.

Factorial 3

Let's run through an entire program with our complete version of factorial.



Required cards

cards	count
false	1
true	1
6	1
etfi	1
ifte	1
≡	1
pop	1
fact	1
—	1
2	2
3	2
dup	2
0	3
1	3
*	4

step	program	data	proc	proc-data	ifte-data
0	3 fact				
1	fact	3			
2	dup 1 — fact * pop 1 dup 0 ≡ ifte etfi	3			
3	pop 1 dup 0 ≡ ifte etfi	dup 1 — fact * 3			
4	dup 0 ≡ ifte etfi	pop 1 dup 1 — fact * 3			

5	ifte etfi	dup 0 ≡ pop 1 dup 1 — fact * 3			
6	etfi	dup 0 ≡ pop 1 dup 1 — fact * 3	ifte		
7	etfi	pop 1 dup 1 — fact * 3	ifte	dup 0 ≡	
8	etfi	dup 1 — fact * 3	ifte	pop 1 dup 0 ≡	

9	etfi	3	ifte	dup 1 — fact * pop 1 dup 0 ≡	
10	etfi	3	ifte	dup 1 — fact * pop 1 dup 0 ≡	
11	etfi	3	ifte	pop 1 dup 0 ≡	dup 1 — fact *
12	etfi	3	ifte	dup 0 ≡	pop 1 dup 1 — fact *

13	dup 0 ≡ etfi	3	ifte		pop 1 dup 1 — fact *
----	---	--------------	-----------------	--	---

18	0 ≡ etfi	3	dup	3	pop 1 dup 1 — fact *
19	0 ≡ etfi	3 3	dup		pop 1 dup 1 — fact *
20	0 ≡ etfi	3 3			pop 1 dup 1 — fact *
21	≡ etfi	0 3 3			pop 1 dup 1 — fact *
22	etfi	0 3 3	≡		pop 1 dup 1 — fact *

23	etfi	3 3	≡	0	pop 1 dup 1 — fact *
24	etfi	3	≡	3 0	pop 1 dup 1 — fact *
25	etfi	3	≡	false	pop 1 dup 1 — fact *
26	etfi	false 3	≡		pop 1 dup 1 — fact *
27	etfi	false 3			pop 1 dup 1 — fact *

28		false 3	etfi		pop 1 dup 1 — fact *
29		3	etfi	false	pop 1 dup 1 — fact *
30	dup 1 — fact *	3	etfi		
31	dup 1 — fact *	3			
32	1 — fact *	3	dup		
33	1 — fact *		dup	3	
34	1 — fact *		dup	3 3	
35	1 — fact *	3	dup	3	

36	1 — fact *	3 3	dup		
37	1 — fact *	3 3			
38	— fact *	1 3 3			
39	fact *	1 3 3	—		
40	fact *	3 3	—	1	
41	fact *	3	—	3 1	
42	fact *	3	—	2	
43	fact *	2 3	—		
44	fact *	2 3			
45	dup 1 — fact * pop 1 dup 0 ≡ ifte etfi *	2 3			

46	pop 1 dup 0 ≡ ifte etfi *	dup 1 — fact * 2 3			
47	dup 0 ≡ ifte etfi *	pop 1 dup 1 — fact * 2 3			
48	ifte etfi *	dup 0 ≡ pop 1 dup 1 — fact * 2 3			

49	etfi *	dup 0 ≡ pop 1 dup 1 — fact * 2 3	ifte		
50	etfi *	pop 1 dup 1 — fact * 2 3	ifte	dup 0 ≡	
51	etfi *	dup 1 — fact * 2 3	ifte	pop 1 dup 0 ≡	
52	etfi *	2 3	ifte	dup 1 — fact * pop 1 dup 0 ≡	

53	etfi *	2 3	ifte	dup 1 — fact * pop 1 dup 0 ≡	
54	etfi *	2 3	ifte	pop 1 dup 0 ≡	dup 1 — fact *
55	etfi *	2 3	ifte	dup 0 ≡	pop 1 dup 1 — fact *
56	dup 0 ≡ etfi *	2 3	ifte		pop 1 dup 1 — fact *

57	dup 0 ≡ etfi *	2 3			pop 1 dup 1 — fact *
58	0 ≡ etfi *	2 3	dup		pop 1 dup 1 — fact *
59	0 ≡ etfi *	3	dup	2	pop 1 dup 1 — fact *
60	0 ≡ etfi *	3	dup	2 2	pop 1 dup 1 — fact *
61	0 ≡ etfi *	2 3	dup	2	pop 1 dup 1 — fact *

62	0 ≡ etfi *	2 2 3	dup		pop 1 dup 1 — fact *
63	0 ≡ etfi *	2 2 3			pop 1 dup 1 — fact *
64	≡ etfi *	0 2 2 3			pop 1 dup 1 — fact *
65	etfi *	0 2 2 3	≡		pop 1 dup 1 — fact *
66	etfi *	2 2 3	≡	0	pop 1 dup 1 — fact *

67	<table><tr><td>etfi</td></tr><tr><td>*</td></tr></table>	etfi	*	<table><tr><td>2</td></tr><tr><td>3</td></tr></table>	2	3	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>2</td></tr><tr><td>0</td></tr></table>	2	0	<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*
etfi																				
*																				
2																				
3																				
≡																				
2																				
0																				
pop	1																			
dup	1																			
—																				
fact	*																			
68	<table><tr><td>etfi</td></tr><tr><td>*</td></tr></table>	etfi	*	<table><tr><td>2</td></tr><tr><td>3</td></tr></table>	2	3	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>false</td></tr></table>	false	<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*	
etfi																				
*																				
2																				
3																				
≡																				
false																				
pop	1																			
dup	1																			
—																				
fact	*																			
69	<table><tr><td>etfi</td></tr><tr><td>*</td></tr></table>	etfi	*	<table><tr><td>false</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	false	2	3	<table><tr><td>≡</td></tr></table>	≡		<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*	
etfi																				
*																				
false																				
2																				
3																				
≡																				
pop	1																			
dup	1																			
—																				
fact	*																			
70	<table><tr><td>etfi</td></tr><tr><td>*</td></tr></table>	etfi	*	<table><tr><td>false</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	false	2	3			<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*		
etfi																				
*																				
false																				
2																				
3																				
pop	1																			
dup	1																			
—																				
fact	*																			
71	<table><tr><td>*</td></tr></table>	*	<table><tr><td>false</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	false	2	3	<table><tr><td>etfi</td></tr></table>	etfi		<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*		
*																				
false																				
2																				
3																				
etfi																				
pop	1																			
dup	1																			
—																				
fact	*																			

72	*	2 3	etfi	false	pop 1 dup 1 — fact *
73	dup 1 — fact * *	2 3	etfi		
74	dup 1 — fact * *	2 3			
75	1 — fact * *	2 3	dup		
76	1 — fact * *	3	dup	2	
77	1 — fact * *	3	dup	2 2	
78	1 — fact * *	2 3	dup	2	

79	1 — fact * *	2 2 3	dup		
80	1 — fact * *	2 2 3			
81	— fact * *	1 2 2 3			
82	fact * *	1 2 2 3	—		
83	fact * *	2 2 3	—	1	
84	fact * *	2 3	—	2 1	
85	fact * *	2 3	—	1	
86	fact * *	1 2 3	—		
87	fact * *	1 2 3			

88	dup 1 — fact * pop 1 dup 0 ≡ ifte etfi * *
----	--

1

2

3

91	ifte etfi * *	dup 0 ≡ pop 1 dup 1 — fact * 1 2 3			
92	etfi * *	dup 0 ≡ pop 1 dup 1 — fact * 1 2 3	ifte		
93	etfi * *	pop 1 dup 1 — fact * 1 2 3	ifte	dup 0 ≡	

94	etfi * *	dup 1 — fact * 1 2 3	ifte	pop 1 dup 0 ≡	
95	etfi * *	1 2 3	ifte	dup 1 — fact * pop 1 dup 0 ≡	
96	etfi * *	1 2 3	ifte	dup 1 — fact * pop 1 dup 0 ≡	
97	etfi * *	1 2 3	ifte	pop 1 dup 0 ≡	dup 1 — fact *

98	etfi * *	1 2 3	ifte	dup 0 ≡	pop 1 dup 1 — fact *
99	dup 0 ≡ etfi * *	1 2 3	ifte		pop 1 dup 1 — fact *
100	dup 0 ≡ etfi * *	1 2 3			pop 1 dup 1 — fact *
101	0 ≡ etfi * *	1 2 3	dup		pop 1 dup 1 — fact *
102	0 ≡ etfi * *	2 3	dup	1	pop 1 dup 1 — fact *

103	0 ≡ etfi * *	2 3	dup	1 1	pop 1 dup 1 — fact *
104	0 ≡ etfi * *	1 2 3	dup	1	pop 1 dup 1 — fact *
105	0 ≡ etfi * *	1 1 2 3	dup		pop 1 dup 1 — fact *
106	0 ≡ etfi * *	1 1 2 3			pop 1 dup 1 — fact *
107	≡ etfi * *	0 1 1 2 3			pop 1 dup 1 — fact *

108	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	<table><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	0	1	1	2	3	<table><tr><td>≡</td></tr></table>	≡		<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*
etfi																						
*																						
*																						
0																						
1																						
1																						
2																						
3																						
≡																						
pop	1																					
dup	1																					
—																						
fact	*																					
109	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	<table><tr><td>1</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	1	1	2	3	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>0</td></tr></table>	0	<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*
etfi																						
*																						
*																						
1																						
1																						
2																						
3																						
≡																						
0																						
pop	1																					
dup	1																					
—																						
fact	*																					
110	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	<table><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	1	2	3	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>1</td></tr><tr><td>0</td></tr></table>	1	0	<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*
etfi																						
*																						
*																						
1																						
2																						
3																						
≡																						
1																						
0																						
pop	1																					
dup	1																					
—																						
fact	*																					
111	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	<table><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	1	2	3	<table><tr><td>≡</td></tr></table>	≡	<table><tr><td>false</td></tr></table>	false	<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*	
etfi																						
*																						
*																						
1																						
2																						
3																						
≡																						
false																						
pop	1																					
dup	1																					
—																						
fact	*																					
112	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	<table><tr><td>false</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	false	1	2	3	<table><tr><td>≡</td></tr></table>	≡		<table><tr><td>pop</td><td>1</td></tr><tr><td>dup</td><td>1</td></tr><tr><td>—</td><td></td></tr><tr><td>fact</td><td>*</td></tr></table>	pop	1	dup	1	—		fact	*	
etfi																						
*																						
*																						
false																						
1																						
2																						
3																						
≡																						
pop	1																					
dup	1																					
—																						
fact	*																					

113	etfi * *	false 1 2 3			pop 1 dup 1 — fact *
114	* *	false 1 2 3	etfi		pop 1 dup 1 — fact *
115	* *	1 2 3	etfi	false	pop 1 dup 1 — fact *
116	dup 1 — fact * * *	1 2 3	etfi		
117	dup 1 — fact * * *	1 2 3			

118	1 — fact * * *	1 2 3	dup		
119	1 — fact * * *	2 3	dup	1	
120	1 — fact * * *	2 3	dup	1 1	
121	1 — fact * * *	1 2 3	dup	1	
122	1 — fact * * *	1 1 2 3	dup		
123	1 — fact * * *	1 1 2 3			
124	— fact * * *	1 1 1 2 3			
125	fact * * *	1 1 1 2 3	—		

126	fact * * *	1 1 2 3	—	1	
127	fact * * *	1 2 3	—	1 1	
128	fact * * *	1 2 3	—	0	
129	fact * * *	0 1 2 3	—		
130	fact * * *	0 1 2 3			
131	dup 1 — fact * pop 1 dup 0 ≡ ifte etfi * * *	0 1 2 3			

132	pop 1 dup 0 ≡ ifte etfi * * *	dup 1 — fact * 0 1 2 3			
133	dup 0 ≡ ifte etfi * * *	pop 1 dup 1 — fact * 0 1 2 3			
134	ifte etfi * * *	dup 0 ≡ pop 1 dup 1 — fact * 0 1 2 3			

135	etfi * * *	dup 0 ≡ pop 1 dup 1 — fact * 0 1 2 3	ifte		
136	etfi * * *	pop 1 dup 1 — fact * 0 1 2 3	ifte	dup 0 ≡	
137	etfi * * *	dup 1 — fact * 0 1 2 3	ifte	pop 1 dup 0 ≡	

138	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	*	<table><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	0	1	2	3	<table><tr><td>ifte</td></tr></table>	ifte	<table><tr><td><table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table></td></tr><tr><td><table><tr><td>pop</td></tr><tr><td>1</td></tr></table></td></tr><tr><td><table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table></td></tr></table>	<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*	<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1	<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡	
etfi																											
*																											
*																											
*																											
0																											
1																											
2																											
3																											
ifte																											
<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*																						
dup																											
1																											
—																											
fact																											
*																											
<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1																									
pop																											
1																											
<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡																								
dup																											
0																											
≡																											
139	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	*	<table><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	0	1	2	3	<table><tr><td>ifte</td></tr></table>	ifte	<table><tr><td><table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table></td></tr><tr><td><table><tr><td>pop</td></tr><tr><td>1</td></tr></table></td></tr><tr><td><table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table></td></tr></table>	<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*	<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1	<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡	
etfi																											
*																											
*																											
*																											
0																											
1																											
2																											
3																											
ifte																											
<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*																						
dup																											
1																											
—																											
fact																											
*																											
<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1																									
pop																											
1																											
<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡																								
dup																											
0																											
≡																											
140	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	*	<table><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	0	1	2	3	<table><tr><td>ifte</td></tr></table>	ifte	<table><tr><td><table><tr><td>pop</td></tr><tr><td>1</td></tr></table></td></tr><tr><td><table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table></td></tr></table>	<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1	<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡	<table><tr><td><table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table></td></tr></table>	<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*
etfi																											
*																											
*																											
*																											
0																											
1																											
2																											
3																											
ifte																											
<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1																									
pop																											
1																											
<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡																								
dup																											
0																											
≡																											
<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*																						
dup																											
1																											
—																											
fact																											
*																											
141	<table><tr><td>etfi</td></tr><tr><td>*</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	etfi	*	*	*	<table><tr><td>0</td></tr><tr><td>1</td></tr><tr><td>2</td></tr><tr><td>3</td></tr></table>	0	1	2	3	<table><tr><td>ifte</td></tr></table>	ifte	<table><tr><td><table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table></td></tr></table>	<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡	<table><tr><td><table><tr><td>pop</td></tr><tr><td>1</td></tr></table></td></tr><tr><td><table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table></td></tr></table>	<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1	<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*
etfi																											
*																											
*																											
*																											
0																											
1																											
2																											
3																											
ifte																											
<table><tr><td>dup</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	dup	0	≡																								
dup																											
0																											
≡																											
<table><tr><td>pop</td></tr><tr><td>1</td></tr></table>	pop	1																									
pop																											
1																											
<table><tr><td>dup</td></tr><tr><td>1</td></tr><tr><td>—</td></tr><tr><td>fact</td></tr><tr><td>*</td></tr></table>	dup	1	—	fact	*																						
dup																											
1																											
—																											
fact																											
*																											

142	dup 0 ≡ etfi * * *	0 1 2 3	ifte		pop 1 dup 1 — fact *
143	dup 0 ≡ etfi * * *	0 1 2 3			pop 1 dup 1 — fact *
144	0 ≡ etfi * * *	0 1 2 3	dup		pop 1 dup 1 — fact *
145	0 ≡ etfi * * *	1 2 3	dup	0	pop 1 dup 1 — fact *
146	0 ≡ etfi * * *	1 2 3	dup	0 0	pop 1 dup 1 — fact *

147	0 ≡ etfi * * *	0 1 2 3	dup	0	pop 1 dup 1 — fact *
148	0 ≡ etfi * * *	0 0 1 2 3	dup		pop 1 dup 1 — fact *
149	0 ≡ etfi * * *	0 0 1 2 3			pop 1 dup 1 — fact *
150	≡ etfi * * *	0 0 0 1 2 3			pop 1 dup 1 — fact *
151	etfi * * *	0 0 0 1 2 3	≡		pop 1 dup 1 — fact *

152	etfi * * *	0 0 1 2 3	≡	0	pop 1 dup 1 — fact *
153	etfi * * *	0 1 2 3	≡	0 0	pop 1 dup 1 — fact *
154	etfi * * *	0 1 2 3	≡	true	pop 1 dup 1 — fact *
155	etfi * * *	true 0 1 2 3	≡		pop 1 dup 1 — fact *
156	etfi * * *	true 0 1 2 3			pop 1 dup 1 — fact *

157	* * *	true 0 1 2 3	etfi		pop 1 dup 1 — fact *
158	* * *	0 1 2 3	etfi	true	pop 1 dup 1 — fact *
159	pop 1 * * *	0 1 2 3	etfi		
160	pop 1 * * *	0 1 2 3			
161	1 * * *	0 1 2 3	pop		
162	1 * * *	1 2 3	pop	0	
163	1 * * *	1 2 3	pop		
164	1 * * *	1 2 3			

165	* * *	1 1 2 3			
166	* *	1 1 2 3	*		
167	* *	1 2 3	*	1	
168	* *	2 3	*	1 1	
169	* *	2 3	*	1	
170	* *	1 2 3	*		
171	* *	1 2 3			
172	*	1 2 3	*		
173	*	2 3	*	1	
174	*	3	*	2 1	
175	*	3	*	2	
176	*	2 3	*		
177	*	2 3			
178		2 3	*		
179		3	*	2	
180			*	3 2	
181			*	6	

182		6	*		
183		6			

Puzzle 5.1

Create a function that takes a number and counts down to zero.

Understanding Recursion

If you had trouble understanding this chapter, go back to the beginning; otherwise, continue to the next chapter.

Chapter 6

Lists

Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

– Linus Torvalds

Organizing Data

In the physical world we have closets, cabinets, desks, and boxes to organize our things. In the programming world we have data structures to organize our data. Just as there are different kinds of organizing structures in the physical world, there are all kinds of data structures in the programming world. In the Programming Cards game (or at least in this book), we aren't going to explore all the different kinds of data structures. Instead, we will focus on what I think is the most important data structure... The list.

Let's first think through why it is essential to organize our data. Imagine you are a chef in a kitchen, but the kitchen has no fridge, counters, cupboards, or drawers. You have all the food and tools to prepare the food in a handful of piles. The stove, the knives, the fruit and vegetables - everything is in a pile. You don't know which pile everything is in but be assured that everything you need is somewhere. How easy do you think it would be to cook in that kitchen?

Now imagine that someone else is the chef and it is your job to give instructions to that person. This is even more difficult than being the chef yourself. When *you* were the chef you had the benefit of being able to do whatever you needed to get the job done. When someone else is the chef you not only have to figure everything out, but you also have to figure out how to explain it to the other person.

Now imagine an organized kitchen. Food is sorted in the fridge, knives are in the knife drawer, and bowls and plates are in the cupboard. Everything is labeled so that you can see where everything is. How much easier is it to be a chef in this kitchen? Now, think how much easier it would be to explain to another chef how to cook a meal. Explaining to someone how to do something is hard enough. If we don't stay organized, things are even more challenging. In programming we care about structuring our data because it makes things easier to explain to the computer and, more importantly, to other programmers.

Lists in The Programming Cards Game

You already have experience with the list data structure. The stack and the quoted program are both special cases of a list.

program stack: List that keeps instructions to the computer organized.

data stack: List that keeps results of commands organized.

ifte-data stack: List that keeps conditional programs organized.

The quoted program is like a mini-program stack that we can move around and give as input to other functions. A list is no different than a quoted program. We will use the rubber band to enclose a list and use the same  to display a list in this book. So why do we call one a quoted program and one a list? We call a sequence of instructions that we want the computer to execute a quoted program, and we call a sequence of data a list, but even so, a quoted program is a list, and a list is a quoted program. They are synonyms.

Up to this point, when we worked with quoted programs, we put all the cards in a rubber band at the beginning of our program. We need to learn how to have the computer create lists on the fly to write more powerful programs and stay organized. All the results of the commands we give the computer end up on one single **data** stack. Being able to create lists in the middle of our program is like being able to use as many **data** stacks as we want. Each list is like it's own mini **data** stack.

Working With Lists

Cons

`cons` takes any type and a list and prepends the card to the list. In other words, cons puts a card on top of the stack in the rubber band.

Function	Input	Output
<code>cons</code>	(Any, List)	(List)

Required cards

cards	count
<code>cons</code>	1
<code>"a"</code>	1

cons example: 1

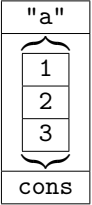
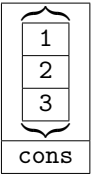
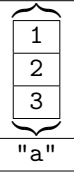
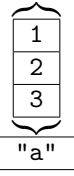
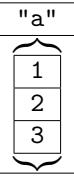
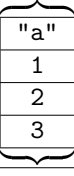
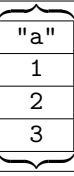
step	program	data	proc	proc-data
0	"a" cons			
1	cons	"a"		
2	cons	"a"		
3		"a"	cons	
4		"a"	cons	
5			cons	"a"
6			cons	"a"
7		"a"	cons	
8		"a"		

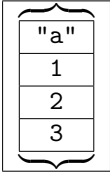
Required cards

cards	count
cons	1
3	1
2	1
1	1
"a"	1

cons example: 2

step	program	data	proc	proc-data
------	---------	------	------	-----------

0				
1				
2				
3				
4				
5				
6				
7				

8				
---	--	---	--	--

Uncons

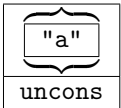
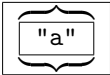
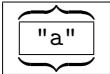
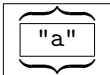
`uncons` is the reverse of `cons`. It takes a list, removes the top card from the list and places the card on top of the list.

Function	Input	Output
<code>uncons</code>	(List)	(Any, List)

Required cards

cards	count
<code>uncons</code>	1
"a"	1

`uncons` example: 1

step	program	data	proc	proc-data
0				
1	<code>uncons</code>			
2			<code>uncons</code>	
3			<code>uncons</code>	
4			<code>uncons</code>	
5		"a"	<code>uncons</code>	

6		"a"	uncons	
7		"a"		

Required cards

cards	count
cons	1
3	1
2	1
1	1
"a"	1

uncons example: 2

step	program	data	proc	proc-data
0	"a" 1 2 3 uncons			
1	uncons	"a" 1 2 3		
2		"a" 1 2 3	uncons	
3			uncons	"a" 1 2 3

4			uncons	"a" 1 2 3
5		"a"	uncons	1 2 3
6		1 2 3 "a"	uncons	
7		1 2 3 "a"		

Null

`null` checks if a list is empty. If the list is empty, it outputs `true`. If the list is not empty, it will output `false`.

Required cards

cards	count
3	1
2	1
1	1
null	1
false	1

null example: 1

step	program	data	proc	proc-data
------	---------	------	------	-----------

0	1 2 3 null			
1	null	1 2 3		
2		1 2 3	null	
3			null	1 2 3
4			null	false
5		false	null	
6		false		

Required cards

cards	count
null	1
true	1

null example: 2

step	program	data	proc	proc-data
0	 null			
1	null			
2			null	
3			null	

4			null	true
5		true	null	
6		true		

Reversing a List

Learning how to reverse a list is good practice for working with and understanding lists. When we work with lists, we need to be able to break them apart and get at what’s inside them. We also need to know how to put a list together to keep things organized. Creating a program that reverses a list does both of these things.

First, let’s play with `cons` and `uncons` to ensure we have a good idea of what is happening.

Required cards

cards	count
3	1
2	1
1	1
cons	3

step	program	data	proc	proc-data
0	1 2 3 cons cons cons			
1	2 3 cons cons cons	1		

2	3 cons cons cons	2 1		
3	cons cons cons	3 2 1		
4	cons cons cons	cons 3 2 1		
5	cons cons	cons 3 2 1	cons	
6	cons cons	3 2 1	cons	cons
7	cons cons	2 1	cons	3 cons
8	cons cons	2 1	cons	cons 3
9	cons cons	cons 3 2 1	cons	
10	cons cons	cons 3 2 1		
11	cons	cons 3 2 1	cons	
12	cons	2 1	cons	cons 3

13	cons	1	cons	2 ┌─┴─┐ 3
14	cons	1	cons	┌─┴─┐ 2 ┌─┴─┐ 3
15	cons	2 ┌─┴─┐ 3 ┌─┴─┐ 1	cons	
16	cons	2 ┌─┴─┐ 3 ┌─┴─┐ 1		
17		2 ┌─┴─┐ 3 ┌─┴─┐ 1	cons	
18		1	cons	2 ┌─┴─┐ 3
19			cons	1 ┌─┴─┐ 2 ┌─┴─┐ 3
20			cons	┌─┴─┐ 1 ┌─┴─┐ 2 ┌─┴─┐ 3
21		┌─┴─┐ 1 ┌─┴─┐ 2 ┌─┴─┐ 3	cons	
22		┌─┴─┐ 1 ┌─┴─┐ 2 ┌─┴─┐ 3		

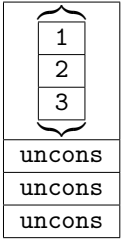
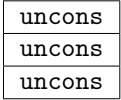
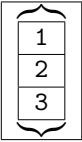
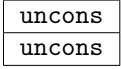
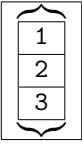
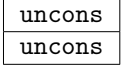
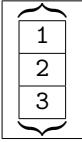
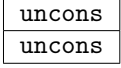
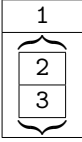
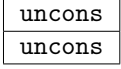
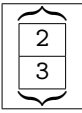
Can you explain in your own words what that program did? First, we told the computer to put some numbers and an empty rubber band on the **data** stack, and then we told the computer to cons the numbers into the rubber band. Notice the order of the number cards. On the data stack they were in one order and when they were in the final list they were

reversed. Lists are like portable stacks. Just like working with a stack we can only ever work with the top card. The first card we put in the rubber band is the bottom of the list. And the most recent card added to the rubber band is the top of the list.

Let's play with `uncons` and see how it undoes `cons`.

Required cards

cards	count
<code>1</code>	1
<code>2</code>	1
<code>3</code>	1
<code>uncons</code>	3

step	program	data	proc	proc-data
0				
1				
2			<code>uncons</code>	
3			<code>uncons</code>	
4			<code>uncons</code>	
5		<code>1</code>	<code>uncons</code>	

6	uncons uncons	2 3 1	uncons	
7	uncons uncons	2 3 1		
8	uncons	2 3 1	uncons	
9	uncons	1	uncons	2 3
10	uncons	1	uncons	2 3
11	uncons	2 1	uncons	3
12	uncons	3 2 1	uncons	
13	uncons	3 2 1		
14		3 2 1	uncons	
15		2 1	uncons	3
16		2 1	uncons	3
17		3 2 1	uncons	

18		 3 2 1	uncons	
19		 3 2 1		

First, we put a list on the stack and then we deconstruct it by running `uncons` three times because there are three elements in the list. Once again, as the list is deconstructed, the order of the cards is reversed.

So far we only wrote functions that work with either the top card or the second from the top card. We simply use `swap` to work with the second from the top. There are times when we need to reach down deeper into the stack and work with the third card. For this exercise, we will work with `rollup` and `rotate`. Make sure you are familiar with those functions before moving on.

Create the Reverse Function

Now let's build it! We are going to go through the function design steps. We will go through each step together, but finding the solution yourself is possible. Here is a hint if you want to try it. The only cards you need are `dup` `swap` `null` `uncons` `cons` `rotate` `rollup` `ifte` `etfi` `pop`. You may use some of those cards more than once, but you can create a reverse function using only those commands.

Step 1: Reverse Type

Function	Input	Output
<code>reverse</code>	(List)	(List)

Step 2: Reverse Examples

Inputs	Outputs
{}	{}
{1}	{1}
{1, 2, 3}	{3, 2, 1}

Step 3: Create the Function Body

What is a simple way to explain the steps to reverse a list to the computer? A trick when working with lists and recursion is using an accumulator. Accumulate means to collect or pile up. We could give the computer instructions to shuffle the cards in the same rubber

band in a way that reverses it, but using the same rubber band gets tricky. So, instead of figuring out how to shuffle cards around in the same rubber band, we use an extra rubber band as the accumulating list.

Here is a way to describe what we want to do in English: Take a list that you want to reverse and an empty list (your accumulator). Take the first card out of the first list and put it at the front of the accumulating list. Repeat until the first list is empty. When the first list is empty, throw it away. The accumulator list will be the reverse of your original list.

If we use an accumulator then our function type needs to change. We need a function that takes two lists as input, not just one. At each recursive step we need the computer to work with both the original and accumulating lists. A recursive function must be connectable with itself, so the two lists must remain in the same order for both the input and output. That means if the function expects the accumulator list first and the original list second as *input*, then it needs to *output* the accumulator list first, and the original list second.

Step 1: Reverse2 Type

Function	Input	Output
<code>reverse2</code>	(List, List)	(List, List)

Step 2: Reverse2 Examples

We will decide that the first list is the list to be reversed, and the second is the accumulating list. You can do it the other way, too, but that is an exercise for the reader.

Inputs	Outputs
{ } { }	{ } { }
{1} { }	{ } {1}
{1 2 3} { }	{ } {3 2 1}

1. Choosing recursion or base case

We are starting with our condition program to determine if we stop or repeat. Since we decided that the original list would be the first input and the accumulating list would be the second input, the original list would be second from the top on the **data** stack. In our recursive functions, we tested the top card of the stack before. How do we test the second from the top card? This is where rollup comes in. By using `dup` `swap` and `rollup` we can duplicate the second from the top card, and it will be on the top of the stack, ready for our condition program to test.

Required cards

cards	count
<code>swap</code>	1
<code>dup</code>	1

rollup	1
"accum"	1
"original"	2

step	program	data	proc	proc-data
0	"original" "accum" swap dup rollup			
1	"accum" swap dup rollup	"original"		
2	swap dup rollup	"accum" "original"		
3	dup rollup	"accum" "original"	swap	
4	dup rollup	"original"	swap	"accum"
5	dup rollup		swap	"original" "accum"
6	dup rollup		swap	"accum" "original"
7	dup rollup	"accum"	swap	"original"
8	dup rollup	"original" "accum"	swap	
9	dup rollup	"original" "accum"		
10	rollup	"original" "accum"	dup	
11	rollup	"accum"	dup	"original"
12	rollup	"accum"	dup	"original" "original"

13	rollup	"original" "accum"	dup	"original"
14	rollup	"original" "original" "accum"	dup	
15	rollup	"original" "original" "accum"		
16		"original" "original" "accum"	rollup	
17		"original" "accum"	rollup	"original"
18		"accum"	rollup	"original" "original"
19			rollup	"accum" "original" "original"
20			rollup	"original" "accum" "original"
21		"original"	rollup	"accum" "original"
22		"accum" "original"	rollup	"original"
23		"original" "accum" "original"	rollup	
24		"original" "accum" "original"		

This is a valuable pattern for getting the second from the last card onto the top of the stack to test it. What should we test it for, though? We want to check if the original list is empty. If it is empty, then we can stop; if it isn't empty, then we still need to fully reverse it.

reverse2	
	swap dup rollup null ? ? ifte etfi

We do `swap` `dup` `rollup` to get the original list to the top, then we do `null` to test it to see if it is empty or not.

2. Base Case

Looking at the examples table, we see that the base case is an empty list. Since the original and accumulating lists are already in the correct order on the **data** stack, the base case is to do nothing.

reverse2	
	swap dup rollup null ? ifte etfi

3. Recursive Case

We want the program to take the top card of one list and put it into the other. Remember that we decided the original will be the first input. That means that it will be second from the top on the **data** stack. The accumulating list will be at the top of the **data** stack. Also, remember that cons and uncons work on the top list, so we will have to use swap and rotate to change the order of the lists. Start by getting the top card from the original list and moving it to the accumulating list.

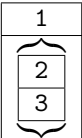
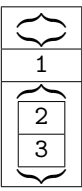
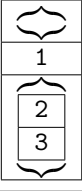
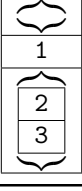
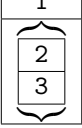
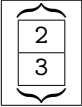
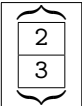
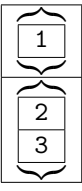
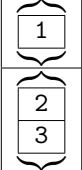
Required cards

cards	count
swap	1
uncons	1
rotate	1
cons	1
3	1
2	1
1	1

step	program	data	proc	proc-data
0	1 2 3 swap uncons rotate cons			
1	swap uncons rotate cons	1 2 3		
2	swap uncons rotate cons	1 2 3		
3	uncons rotate cons	1 2 3	swap	

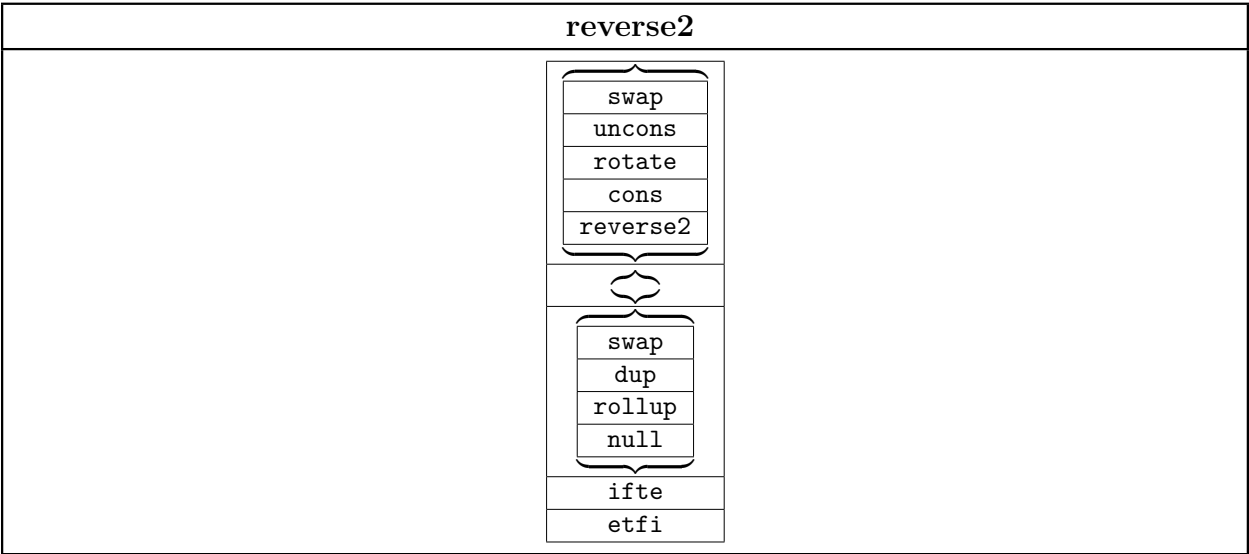
4	uncons rotate cons	1 2 3	swap	
5	uncons rotate cons		swap	1 2 3
6	uncons rotate cons		swap	1 2 3
7	uncons rotate cons		swap	1 2 3
8	uncons rotate cons	1 2 3	swap	
9	uncons rotate cons	1 2 3		
10	rotate cons	1 2 3	uncons	
11	rotate cons		uncons	1 2 3
12	rotate cons		uncons	1 2 3

13	rotate cons	1	uncons	2 3
14	rotate cons	2 3 1	uncons	
15	rotate cons	2 3 1		
16	cons	2 3 1	rotate	
17	cons	1	rotate	2 3
18	cons		rotate	1 2 3
19	cons		rotate	1 2 3
20	cons		rotate	2 3 1
21	cons	2 3	rotate	1

22	<code>cons</code>		<code>rotate</code>	
23	<code>cons</code>		<code>rotate</code>	
24	<code>cons</code>			
25			<code>cons</code>	
26			<code>cons</code>	
27			<code>cons</code>	
28			<code>cons</code>	
29			<code>cons</code>	
30				

That works! Also, it leaves the lists in the correct order on the stack, but we need to

repeat for it to be the recursive case. So all we do is add `reverse2` at the end. The final function looks like this.



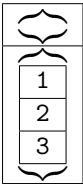
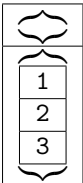
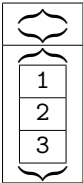
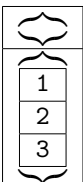
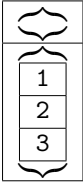
Required cards

cards	count
false	1
true	1
etfi	1
ifte	1
null	1
rollup	1
dup	1
reverse2	1
cons	1
rotate	1
uncons	1
1	2
2	2
3	2
swap	2

step	program	data	proc	proc-data	ifte-data
0	1 2 3 reverse2				
1	reverse2	1 2 3			
2	reverse2	1 2 3			
3	swap uncons rotate cons reverse2 swap dup rollup null ifte etfi	1 2 3			
4	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 1 2 3			

5	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 1 2 3			
6	ifte etfi	swap dup rollup null swap uncons rotate cons reverse2 1 2 3			
7	etfi	swap dup rollup null swap uncons rotate cons reverse2 1 2 3	ifte		

8	etfi	swap uncons rotate cons reverse2 1 2 3	ifte	swap dup rollup null	
9	etfi	swap uncons rotate cons reverse2 1 2 3	ifte	swap dup rollup null	
10	etfi	1 2 3	ifte	swap uncons rotate cons reverse2 swap dup rollup null	

11	etfi		ifte	swap uncons rotate cons reverse2 swap dup rollup null	
12	etfi		ifte	swap dup rollup null swap uncons rotate cons reverse2	
13	etfi		ifte	swap dup rollup null swap uncons rotate cons reverse2	
14	swap dup rollup null etfi		ifte		swap uncons rotate cons reverse2
15	swap dup rollup null etfi				swap uncons rotate cons reverse2

16	dup rollup null etfi	1 2 3	swap		swap uncons rotate cons reverse2
17	dup rollup null etfi	1 2 3	swap		swap uncons rotate cons reverse2
18	dup rollup null etfi		swap	1 2 3	swap uncons rotate cons reverse2
19	dup rollup null etfi		swap	1 2 3	swap uncons rotate cons reverse2
20	dup rollup null etfi		swap	1 2 3	swap uncons rotate cons reverse2
21	dup rollup null etfi	1 2 3	swap		swap uncons rotate cons reverse2

22	dup rollup null etfi	1 2 3			swap uncons rotate cons reverse2
23	rollup null etfi	1 2 3	dup		swap uncons rotate cons reverse2
24	rollup null etfi		dup	1 2 3	swap uncons rotate cons reverse2
25	rollup null etfi		dup	1 2 3 1 2 3	swap uncons rotate cons reverse2
26	rollup null etfi	1 2 3	dup	1 2 3	swap uncons rotate cons reverse2

27	rollup null etfi	1 2 3 1 2 3	dup		swap uncons rotate cons reverse2
28	rollup null etfi	1 2 3 1 2 3			swap uncons rotate cons reverse2
29	null etfi	1 2 3 1 2 3	rollup		swap uncons rotate cons reverse2
30	null etfi	1 2 3	rollup	1 2 3	swap uncons rotate cons reverse2
31	null etfi		rollup	1 2 3 1 2 3	swap uncons rotate cons reverse2

32	null etfi		rollup	1 2 3 1 2 3	swap uncons rotate cons reverse2
33	null etfi		rollup	1 2 3 1 2 3	swap uncons rotate cons reverse2
34	null etfi	1 2 3	rollup	1 2 3	swap uncons rotate cons reverse2
35	null etfi	1 2 3	rollup	1 2 3	swap uncons rotate cons reverse2
36	null etfi	1 2 3 1 2 3	rollup		swap uncons rotate cons reverse2

37	null etfi	1 2 3 1 2 3			swap uncons rotate cons reverse2
38	etfi	1 2 3 1 2 3	null		swap uncons rotate cons reverse2
39	etfi	1 2 3	null	1 2 3	swap uncons rotate cons reverse2
40	etfi	1 2 3	null	false	swap uncons rotate cons reverse2
41	etfi	false 1 2 3	null		swap uncons rotate cons reverse2

42	etfi	false 1 2 3			swap uncons rotate cons reverse2
43		false 1 2 3	etfi		swap uncons rotate cons reverse2
44		1 2 3	etfi	false	swap uncons rotate cons reverse2
45	swap uncons rotate cons reverse2	1 2 3	etfi		
46	swap uncons rotate cons reverse2	1 2 3			
47	uncons rotate cons reverse2	1 2 3	swap		
48	uncons rotate cons reverse2	1 2 3	swap		

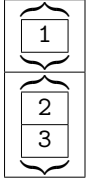
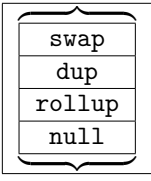
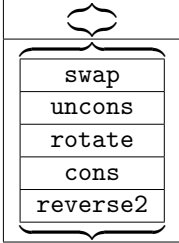
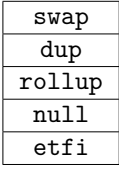
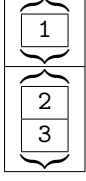
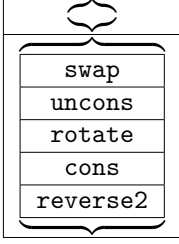
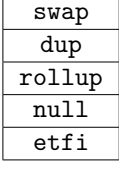
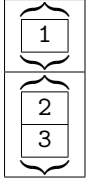
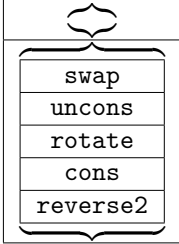
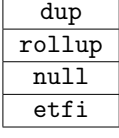
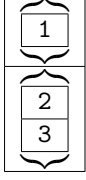
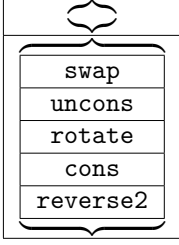
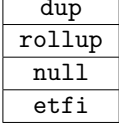
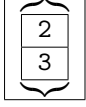
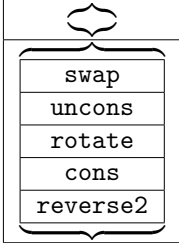
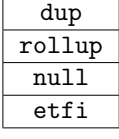
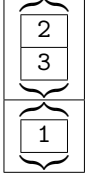
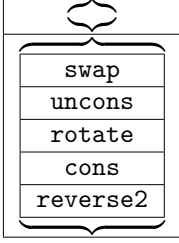
49	uncons rotate cons reverse2		swap	1 2 3	
50	uncons rotate cons reverse2		swap	1 2 3	
51	uncons rotate cons reverse2		swap	1 2 3	
52	uncons rotate cons reverse2	1 2 3	swap		
53	uncons rotate cons reverse2	1 2 3			
54	rotate cons reverse2	1 2 3	uncons		
55	rotate cons reverse2		uncons	1 2 3	
56	rotate cons reverse2		uncons	1 2 3	
57	rotate cons reverse2	1	uncons	2 3	

67	cons reverse2	1 2 3	rotate		
68	cons reverse2	1 2 3			
69	reverse2	1 2 3	cons		
70	reverse2	1 2 3	cons		
71	reverse2	2 3	cons	1	
72	reverse2	2 3	cons	1	
73	reverse2	1 2 3	cons		
74	reverse2	1 2 3			

75	swap uncons rotate cons reverse2 swap dup rollup null ifte etfi	1 2 3			
76	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 1 2 3			
77	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 1 2 3			

78	ifte etfi	swap dup rollup null swap uncons rotate cons reverse2 1 2 3			
79	etfi	swap dup rollup null swap uncons rotate cons reverse2 1 2 3	ifte		
80	etfi	swap uncons rotate cons reverse2 1 2 3	ifte	swap dup rollup null	

81	etfi	swap uncons rotate cons reverse2 1 2 3	ifte	swap dup rollup null	
82	etfi	1 2 3	ifte	swap uncons rotate cons reverse2 swap dup rollup null	
83	etfi	1 2 3	ifte	swap uncons rotate cons reverse2 swap dup rollup null	
84	etfi	1 2 3	ifte	swap dup rollup null	swap uncons rotate cons reverse2

85	etfi		ifte		
86			ifte		
87					
88			swap		
89			swap		
90			swap		

91	dup rollup null etfi		swap	1 2 3	swap uncons rotate cons reverse2
92	dup rollup null etfi	1	swap	2 3	swap uncons rotate cons reverse2
93	dup rollup null etfi	2 3 1	swap		swap uncons rotate cons reverse2
94	dup rollup null etfi	2 3 1			swap uncons rotate cons reverse2
95	rollup null etfi	2 3 1	dup		swap uncons rotate cons reverse2
96	rollup null etfi	1	dup	2 3	swap uncons rotate cons reverse2

97	rollup null etfi	1	dup	2 3 2 3	swap uncons rotate cons reverse2
98	rollup null etfi	2 3 1	dup	2 3	swap uncons rotate cons reverse2
99	rollup null etfi	2 3 2 3 1	dup		swap uncons rotate cons reverse2
100	rollup null etfi	2 3 2 3 1			swap uncons rotate cons reverse2
101	null etfi	2 3 2 3 1	rollup		swap uncons rotate cons reverse2

102	null etfi	2 3 1	rollup	2 3	swap uncons rotate cons reverse2
103	null etfi	1	rollup	2 3 2 3	swap uncons rotate cons reverse2
104	null etfi		rollup	1 2 3 2 3	swap uncons rotate cons reverse2
105	null etfi		rollup	2 3 1 2 3	swap uncons rotate cons reverse2
106	null etfi	2 3	rollup	1 2 3	swap uncons rotate cons reverse2
107	null etfi	1 2 3	rollup	2 3	swap uncons rotate cons reverse2

108	null etfi	2 3 1 2 3	rollup		swap uncons rotate cons reverse2
109	null etfi	2 3 1 2 3			swap uncons rotate cons reverse2
110	etfi	2 3 1 2 3	null		swap uncons rotate cons reverse2
111	etfi	1 2 3	null	2 3	swap uncons rotate cons reverse2
112	etfi	1 2 3	null	false	swap uncons rotate cons reverse2

113	etfi	false 1 2 3	null		swap uncons rotate cons reverse2
114	etfi	false 1 2 3			swap uncons rotate cons reverse2
115		false 1 2 3	etfi		swap uncons rotate cons reverse2
116		1 2 3	etfi	false	swap uncons rotate cons reverse2
117	swap uncons rotate cons reverse2	1 2 3	etfi		
118	swap uncons rotate cons reverse2	1 2 3			
119	uncons rotate cons reverse2	1 2 3	swap		

120	uncons rotate cons reverse2	2 3	swap	1	
121	uncons rotate cons reverse2		swap	2 3 1	
122	uncons rotate cons reverse2		swap	1 2 3	
123	uncons rotate cons reverse2	1	swap	2 3	
124	uncons rotate cons reverse2	2 3 1	swap		
125	uncons rotate cons reverse2	2 3 1			
126	rotate cons reverse2	2 3 1	uncons		
127	rotate cons reverse2	1	uncons	2 3	
128	rotate cons reverse2	1	uncons	2 3	

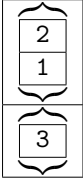
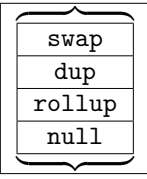
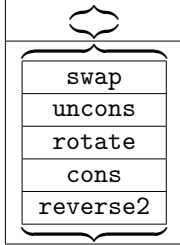
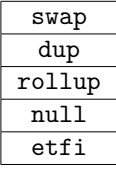
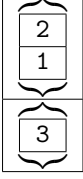
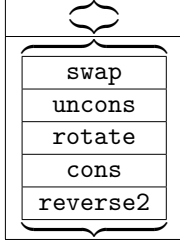
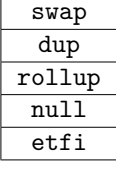
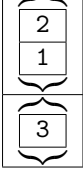
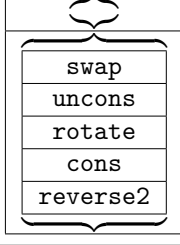
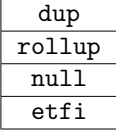
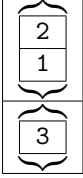
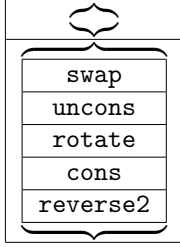
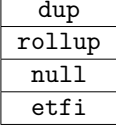
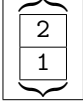
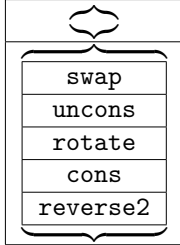
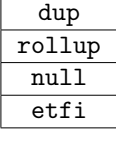
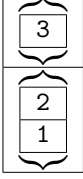
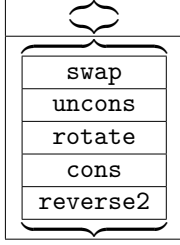
129	rotate cons reverse2	2 1	uncons	3	
130	rotate cons reverse2	3 2 1	uncons		
131	rotate cons reverse2	3 2 1			
132	cons reverse2	3 2 1	rotate		
133	cons reverse2	2 1	rotate	3	
134	cons reverse2	1	rotate	2 3	
135	cons reverse2		rotate	1 2 3	
136	cons reverse2		rotate	3 2 1	
137	cons reverse2	3	rotate	2 1	
138	cons reverse2	2 3	rotate	1	

139	cons reverse2	1 2 3	rotate		
140	cons reverse2	1 2 3			
141	reverse2	1 2 3	cons		
142	reverse2	2 3	cons	1	
143	reverse2	3	cons	2 1	
144	reverse2	3	cons	2 1	
145	reverse2	2 1 3	cons		
146	reverse2	2 1 3			

147	swap uncons rotate cons reverse2 swap dup rollup null ifte etfi	2 1 3			
148	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 2 1 3			
149	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 2 1 3			

150	ifte etfi	swap dup rollup null swap uncons rotate cons reverse2 2 1 3			
151	etfi	swap dup rollup null swap uncons rotate cons reverse2 2 1 3	ifte		
152	etfi	swap uncons rotate cons reverse2 2 1 3	ifte	swap dup rollup null	

153	etfi	swap uncons rotate cons reverse2 2 1 3	ifte	swap dup rollup null	
154	etfi	2 1 3	ifte	swap uncons rotate cons reverse2 swap dup rollup null	
155	etfi	2 1 3	ifte	swap uncons rotate cons reverse2 swap dup rollup null	
156	etfi	2 1 3	ifte	swap dup rollup null	swap uncons rotate cons reverse2

157	etfi		ifte		
158			ifte		
159					
160			swap		
161			swap		
162			swap		

163	dup rollup null etfi		swap	2 1 3	swap uncons rotate cons reverse2
164	dup rollup null etfi	2 1	swap	3	swap uncons rotate cons reverse2
165	dup rollup null etfi	3 2 1	swap		swap uncons rotate cons reverse2
166	dup rollup null etfi	3 2 1			swap uncons rotate cons reverse2
167	rollup null etfi	3 2 1	dup		swap uncons rotate cons reverse2
168	rollup null etfi	2 1	dup	3	swap uncons rotate cons reverse2

169	rollup null etfi	2 1	dup	3 3	swap uncons rotate cons reverse2
170	rollup null etfi	3 2 1	dup	3	swap uncons rotate cons reverse2
171	rollup null etfi	3 3 2 1	dup		swap uncons rotate cons reverse2
172	rollup null etfi	3 3 2 1			swap uncons rotate cons reverse2
173	null etfi	3 3 2 1	rollup		swap uncons rotate cons reverse2
174	null etfi	3 2 1	rollup	3	swap uncons rotate cons reverse2

175	null etfi	2 1	rollup	3 3	swap uncons rotate cons reverse2
176	null etfi		rollup	2 1 3 3	swap uncons rotate cons reverse2
177	null etfi		rollup	3 2 1 3	swap uncons rotate cons reverse2
178	null etfi	3	rollup	2 1 3	swap uncons rotate cons reverse2
179	null etfi	2 1 3	rollup	3	swap uncons rotate cons reverse2
180	null etfi	3 2 1 3	rollup		swap uncons rotate cons reverse2

181	null etfi	3 2 1 3			swap uncons rotate cons reverse2
182	etfi	3 2 1 3	null		swap uncons rotate cons reverse2
183	etfi	2 1 3	null	3	swap uncons rotate cons reverse2
184	etfi	2 1 3	null	false	swap uncons rotate cons reverse2
185	etfi	false 2 1 3	null		swap uncons rotate cons reverse2
186	etfi	false 2 1 3			swap uncons rotate cons reverse2

187		false 2 1 3	etfi		swap uncons rotate cons reverse2
188		2 1 3	etfi	false	swap uncons rotate cons reverse2
189	swap uncons rotate cons reverse2	2 1 3	etfi		
190	swap uncons rotate cons reverse2	2 1 3			
191	uncons rotate cons reverse2	2 1 3	swap		
192	uncons rotate cons reverse2	3	swap	2 1	
193	uncons rotate cons reverse2		swap	3 2 1	

194	uncons rotate cons reverse2		swap	2 1 3	
195	uncons rotate cons reverse2	2 1	swap	3	
196	uncons rotate cons reverse2	3 2 1	swap		
197	uncons rotate cons reverse2	3 2 1			
198	rotate cons reverse2	3 2 1	uncons		
199	rotate cons reverse2	2 1	uncons	3	
200	rotate cons reverse2	2 1	uncons	3	
201	rotate cons reverse2	3 2 1	uncons		
202	rotate cons reverse2	3 2 1	uncons		

203	rotate cons reverse2	3 2 1			
204	cons reverse2	3 2 1	rotate		
205	cons reverse2	3 2 1	rotate		
206	cons reverse2	2 1	rotate	3	
207	cons reverse2		rotate	2 1 3	
208	cons reverse2		rotate	3 2 1	
209	cons reverse2		rotate	3 2 1	
210	cons reverse2	3	rotate	2 1	
211	cons reverse2	2 1 3	rotate		

212	cons reverse2	2 1 3			
213	reverse2	2 1 3	cons		
214	reverse2	3	cons	2 1	
215	reverse2		cons	3 2 1	
216	reverse2		cons	3 2 1	
217	reverse2	3 2 1	cons		
218	reverse2	3 2 1			

219	swap uncons rotate cons reverse2 swap dup rollup null ifte etfi	3 2 1			
220	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 3 2 1			
221	swap dup rollup null ifte etfi	swap uncons rotate cons reverse2 3 2 1			

222	ifte etfi	swap dup rollup null swap uncons rotate cons reverse2 3 2 1			
223	etfi	swap dup rollup null swap uncons rotate cons reverse2 3 2 1	ifte		
224	etfi	swap uncons rotate cons reverse2 3 2 1	ifte	swap dup rollup null	

225	etfi	swap uncons rotate cons reverse2 3 2 1	ifte	swap dup rollup null	
226	etfi	3 2 1	ifte	swap uncons rotate cons reverse2 swap dup rollup null	
227	etfi	3 2 1	ifte	swap uncons rotate cons reverse2 swap dup rollup null	
228	etfi	3 2 1	ifte	swap dup rollup null	swap uncons rotate cons reverse2

229	etfi	3 2 1	ifte	swap dup rollup null	swap uncons rotate cons reverse2
230	swap dup rollup null etfi	3 2 1	ifte		swap uncons rotate cons reverse2
231	swap dup rollup null etfi	3 2 1			swap uncons rotate cons reverse2
232	dup rollup null etfi	3 2 1	swap		swap uncons rotate cons reverse2
233	dup rollup null etfi		swap	3 2 1	swap uncons rotate cons reverse2
234	dup rollup null etfi		swap	3 2 1	swap uncons rotate cons reverse2

235	dup rollup null etfi		swap	3 2 1	swap uncons rotate cons reverse2
236	dup rollup null etfi	3 2 1	swap		swap uncons rotate cons reverse2
237	dup rollup null etfi	3 2 1	swap		swap uncons rotate cons reverse2
238	dup rollup null etfi	3 2 1			swap uncons rotate cons reverse2
239	rollup null etfi	3 2 1	dup		swap uncons rotate cons reverse2
240	rollup null etfi	3 2 1	dup		swap uncons rotate cons reverse2

241	rollup null etfi	3 2 1	dup		swap uncons rotate cons reverse2
242	rollup null etfi	3 2 1	dup		swap uncons rotate cons reverse2
243	rollup null etfi	3 2 1	dup		swap uncons rotate cons reverse2
244	rollup null etfi	3 2 1			swap uncons rotate cons reverse2
245	null etfi	3 2 1	rollup		swap uncons rotate cons reverse2
246	null etfi	3 2 1	rollup		swap uncons rotate cons reverse2

247	null etfi	3 2 1	rollup		swap uncons rotate cons reverse2
248	null etfi		rollup	3 2 1	swap uncons rotate cons reverse2
249	null etfi		rollup	3 2 1	swap uncons rotate cons reverse2
250	null etfi		rollup	3 2 1	swap uncons rotate cons reverse2
251	null etfi	3 2 1	rollup		swap uncons rotate cons reverse2
252	null etfi	3 2 1	rollup		swap uncons rotate cons reverse2

253	null etfi	3 2 1			swap uncons rotate cons reverse2
254	etfi	3 2 1	null		swap uncons rotate cons reverse2
255	etfi	3 2 1	null		swap uncons rotate cons reverse2
256	etfi	3 2 1	null	true	swap uncons rotate cons reverse2
257	etfi	true 3 2 1	null		swap uncons rotate cons reverse2
258	etfi	true 3 2 1			swap uncons rotate cons reverse2

259		true 3 2 1	etfi		swap uncons rotate cons reverse2
260		3 2 1	etfi	true	swap uncons rotate cons reverse2
261		3 2 1	etfi		
262		3 2 1			

We did it! Now, we have a function that can reverse any list. But what if we forget to give an empty list as the accumulating list? That would make it a nonsense program. It isn't very nice to have to remember to put an empty list on the stack before using `reverse2`. Furthermore, having an extra list on the stack to clean up later isn't very nice. We can fix that by creating a wrapping function to put the empty list on the stack and clean it up after running the `reverse2` function.

reverse	
	reverse2 swap pop

Go ahead and try it out on your own.

Puzzle 6.1

Create a function similar to the counting functions in the previous chapter. Only this time, put the results in a list.

Chapter 7

User Input

Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

– Alan Perlis

Power Overwhelming

To create more powerful programs we need to be able to interact with the computer while it is running. Think how useful it would be to have a phone or tablet that didn't respond to anything you did. You swipe your finger across the screen, but it doesn't change what it is doing. This is what the Programming Cards game has been like so far. We write a program for the computer to run, and the computer runs it. If we want the computer to run something different, we have to write a new program and have the computer run it from the beginning. This would be like taking your phone to the repair shop whenever you want to switch apps. We must teach our computer how to take input from a user and how to give output.

So far in the Programming Cards game we have played the roles of programmer and computer. Now, we want to add the user as a third role. The role of the user is to give the computer cards when the computer asks for them.

Read

The `read` command is used to read a single new card (or quoted program) from the user. When the computer runs the process for the `read` command, it waits for a card from the user and places it on the **proc-data** stack. The computer cannot proceed until it receives a card from the user.

Function	Input	Output
<code>read</code>	()	(Any)

Print

The `print` command gives information to the user. `print` takes one card from the **data** stack and moves it to the **user-out** stack.

Function	Input	Output
<code>print</code>	(Any)	(Any)

Read/Print Example

Run the program below that uses the two new commands.

Required cards

cards	count
<code>+</code>	1
<code>print</code>	1
<code>2</code>	1
<code>read</code>	2
<code>1</code>	2

step	user-out	program	data	proc	proc-data
0		read read + print			
1		read + print		read	

The process for the `read` command is running; therefore, we are waiting for the user to give a card to the computer. As the user, place the `1` card on the **proc-data** stack. Then, finish the process by moving the card from the **proc-data** stack to the **data** stack, and discarding the `read` card.

step	user-out	program	data	proc	proc-data
2		read + print		read	1

Now that a card has been placed on the **proc-data** stack, the computer can proceed.

step	user-out	program	data	proc	proc-data
3		read + print	1	read	
4		read + print	1		
5		+ print	1	read	

Now, the computer waits for user to place another 1 card on the **proc-data** stack. As the user, place another 1 card on the **proc-data** stack so the computer can continue.

step	user-out	program	data	proc	proc-data
6		+ print	1	read	1
7		+ print	1 1	read	
8		+ print	1 1		
9		print	1 1	+	
10		print	1	+	1
11		print		+	1 1
12		print		+	2
13		print	2	+	
14		print	2		
15			2	print	
16				print	2
17				print	2
18	2			print	

19	2				
----	---	--	--	--	--

In programming it often takes a few minor tweaks to unleash immense power. The program above takes two cards from the user and adds them together, then outputs the result for the user to see. What would happen if the user provided a string instead of numbers? The computer would crash because that would be a nonsense program. Keep in mind that as we grew in the ability to create more powerful programs, we also grew in the number of problems our programs can have. It is part of the programmer's job to guard against the user giving bad input. In the next section we will learn a little about how to ensure our programs are safer from user input.

Safer User Input

To ensure that the user doesn't supply bad input to our calculator function, we will need to introduce some new functions to the computer. These functions test the type of the card.

string?

`string?` is a function that takes any card as input and outputs true if that card is a string. Otherwise, it outputs false.

Function	Input	Output
<code>string?</code>	(Any)	(Boolean)

Required cards

cards	count
1	1
false	1
"hello"	1
string?	1
true	1

step	program	data	proc	proc-data
0	"hello" string?			

1	string?	"hello"		
2		"hello"	string?	
3			string?	"hello"
4			string?	true
5		true	string?	
6		true		

step	program	data	proc	proc-data
0	1 string?			
1	string?	1		
2		1	string?	
3			string?	1
4			string?	false
5		false	string?	
6		false		

number?

`number?` is a function that takes any card as input and outputs true if that card is a number. Otherwise, it outputs false.

Function	Input	Output
<code>number?</code>	(Any)	(Boolean)

Required cards

cards	count
"hello"	1
false	1
1	1
number?	1

<code>true</code>	1
-------------------	---

step	program	data	proc	proc-data
0	<code>1</code> <code>number?</code>			
1	<code>number?</code>	<code>1</code>		
2		<code>1</code>	<code>number?</code>	
3			<code>number?</code>	<code>1</code>
4			<code>number?</code>	<code>true</code>
5		<code>true</code>	<code>number?</code>	
6		<code>true</code>		

step	program	data	proc	proc-data
0	<code>"hello"</code> <code>number?</code>			
1	<code>number?</code>	<code>"hello"</code>		
2		<code>"hello"</code>	<code>number?</code>	
3			<code>number?</code>	<code>"hello"</code>
4			<code>number?</code>	<code>false</code>
5		<code>false</code>	<code>number?</code>	
6		<code>false</code>		

boolean?

`boolean?` is a function that takes any card as input and outputs true if that card is a boolean. Otherwise, it outputs false.

Function	Input	Output
<code>boolean?</code>	(Any)	(Boolean)

Required cards

cards	count
<code>1</code>	1
<code>false</code>	1
<code>boolean?</code>	1
<code>true</code>	1

step	program	data	proc	proc-data
0	<code>true</code> <code>boolean?</code>			
1	<code>boolean?</code>	<code>true</code>		
2		<code>true</code>	<code>boolean?</code>	
3			<code>boolean?</code>	<code>true</code>
4			<code>boolean?</code>	<code>true</code>
5		<code>true</code>	<code>boolean?</code>	
6		<code>true</code>		

step	program	data	proc	proc-data
0	<code>1</code> <code>boolean?</code>			
1	<code>boolean?</code>	<code>1</code>		
2		<code>1</code>	<code>boolean?</code>	
3			<code>boolean?</code>	<code>1</code>
4			<code>boolean?</code>	<code>false</code>
5		<code>false</code>	<code>boolean?</code>	
6		<code>false</code>		

Puzzle 7.1

Create a function that will read input from the user and check to make sure it is a number. If it isn't a number then it will print a message to the user to ask them for a number. The function should recurse until the user gives a number.

Puzzle 7.2

Create a function that safely takes two number inputs from the user adds them together, and outputs the result to the user.

Chapter 8

FizzBuzz

My goal wasn't to make a ton of money. It was to build good computers.

– Steve Wozniak

The Business of Programming

I once worked with someone who made a good amount of money programming. One day at work he told everyone that his son brought home a paycheck that was bigger than his. His son was writing software for a popular video game and his son's software had become very popular in that gaming community. I would bet that he was a better programmer than his son, but his son found something that he could build with his skill that lots of people wanted. I love telling this story to people because it highlights an essential truth about making money with programming.

Making money with programming, as with anything, means that you possess something of value that someone with money is willing to trade their money for. The person with money would rather have what you possess than their money, and you would rather have their money than what you possess. This is why we call it free trade: You are free to trade what you have, and they are free to trade what they have. To make money with programming is as simple as trading your skill or something you built with someone else.

The wonderful thing about developing programming skills is that it opens up so many possibilities for you to find people to trade with. Even if you don't use your programming skills to write code, you still learn to find solutions to problems others don't know how to solve. The kid who made more money than his dad did so because he found a solution to a problem that more people wanted solved than his dad.

Learning programming will enable you to solve more problems, make the world a better place, and allow you to find more opportunities to trade, but this is only part of the story. You need more than just the skill; you need to convince others that you have the skill. A college degree or certification can help a little, but more is required. To convince other people that you have the skill of programming, you have to show them.

When a software company wants to hire a software engineer (degree or no degree), they will give the engineer a programming test. The test usually involves solving a puzzle by

writing a small program. This allows the engineer to show the company they possess the programming skills it desires. Sometimes, the company doesn't care if the engineer gets the right solution. They care about how the engineer communicated and how they thought about solving the problem. A programmer who can communicate well and think through challenging problems stands out. Let's go through an example puzzle to test ourselves and see if we stack up to software engineers who get paid to do this for a living. Keep in mind that this puzzle is rated at the beginner level, but even so, plenty of beginner engineers have been hired at software companies.

FizzBuzz Puzzle

You need to write a program that prints the numbers from 1 to 100 such that:

- If the number is a multiple of 3, you need to output "fizz" instead of that number.
- If the number is a multiple of 5, you need to output "buzz" instead of that number.
- If the number is a multiple of both 3 and 5, you need to output "fizzbuzz" instead of that number.

To solve this, we will need conditionals, recursion, and the ability to test if one number is a multiple of another. We know how to do the first two, but what about testing for multiples? This is where modular arithmetic comes in.

Modular Arithmetic

Modular arithmetic is being able to do arithmetic with numbers that wrap around. A clock is an example of numbers that wrap around. Once the clock reaches 12 it wraps around to 1. Let's say we want to know where the hour hand will be in 14 hours. We can count 14 numbers on the clock or divide 14 by 12 and count the remainder. $14 / 12 = 1 \text{ rem } 2$. Then we count 2 more hours to get our answer. On the clock, 12 is what we call the modulus. It is the number that ends the cycle and starts the counting back at 1.

Another way to find out where the hour hand will be in 14 hours is to ask what 14 modulo 12 is. When we divide, we either get a whole number or a number with a remainder. Modulo is a function that only gives the remainder. 14 modulo 12 is 2. This modulo function is what we want to use to check if a number is a multiple of another number.

In the Programming Cards game, a human plays as the computer, and humans aren't as good at modular arithmetic as digital computers. Interestingly, even certain types of digital computers are better or worse at certain functions than other digital computers. As programmers we need to be able to work around this sometimes.

One way to accommodate a computer that is slow at computing a particular function is to perform the computations beforehand and save it in a table for the computer to look up. We will do that for our human computer. Let's create a table for all the modulo 12 answers up to 24.

input	Modulo 12 output
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	0
13	1
14	2
15	3
16	4
17	5
18	6
19	7
20	8
21	9
22	10
23	11
24	0

With our table our human computer can quickly compute the correct answers for modulo as long as the modulus is 12.

Function	Input	Output
<code>modulo</code>	(Number, Number)	(Number)

Give the computer this program and start to run it.

Required cards

cards	count
<code>12</code>	1
<code>18</code>	1
<code>modulo</code>	1
<code>6</code>	1

step	program	data	proc	proc-data
0	18 12 modulo			
1	12 modulo	18		
2	modulo	12 18		
3		12 18	modulo	
4		18	modulo	12
5			modulo	18 12

Now, use the table to find the correct answer.

step	program	data	proc	proc-data
6			modulo	6
7		6	modulo	
8		6		

By creating the table for the computer to look up, you saved the computer from needing to use division to find the result.

FizzBuzz Modulo Table

The FizzBuzz puzzle requires that our computer compute modulo 3 and modulo 5. So, we will provide the computer with a table for those inputs.

input	modulo 3	modulo 5
1	1	1
2	2	2
3	0	3
4	1	4

5	2	0
6	0	1
7	1	2
8	2	3
9	0	4
10	1	0
11	2	1
12	0	2
13	1	3
14	2	4
15	0	0
16	1	1
17	2	2
18	0	3
19	1	4
20	2	0
21	0	1
22	1	2
23	2	3
24	0	4
25	1	0
26	2	1
27	0	2
28	1	3
29	2	4
30	0	0
31	1	1
32	2	2
33	0	3
34	1	4
35	2	0
36	0	1
37	1	2
38	2	3
39	0	4
40	1	0
41	2	1

42	0	2
43	1	3
44	2	4
45	0	0
46	1	1
47	2	2
48	0	3
49	1	4
50	2	0
51	0	1
52	1	2
53	2	3
54	0	4
55	1	0
56	2	1
57	0	2
58	1	3
59	2	4
60	0	0
61	1	1
62	2	2
63	0	3
64	1	4
65	2	0
66	0	1
67	1	2
68	2	3
69	0	4
70	1	0
71	2	1
72	0	2
73	1	3
74	2	4
75	0	0
76	1	1
77	2	2
78	0	3

79	1	4
80	2	0
81	0	1
82	1	2
83	2	3
84	0	4
85	1	0
86	2	1
87	0	2
88	1	3
89	2	4
90	0	0
91	1	1
92	2	2
93	0	3
94	1	4
95	2	0
96	0	1
97	1	2
98	2	3
99	0	4
100	1	0

With the table above, let's ensure we understand how to test for multiples. If the modulo of a number is 0 then the number is perfectly divisible. In other words, if the remainder is 0 then the number is divisible. This is how we would use that in a program.

Required cards

cards	count
5	1
15	1
true	1
3	1
10	1
modulo	1
1	1

≡	1
false	1
0	2

step	program	data	proc	proc-data
0	10 3 modulo 0 ≡			
1	3 modulo 0 ≡	10		
2	modulo 0 ≡	3 10		
3	0 ≡	3 10	modulo	
4	0 ≡	10	modulo	3
5	0 ≡		modulo	10 3
6	0 ≡		modulo	1
7	0 ≡	1	modulo	
8	0 ≡	1		
9	≡	0 1		
10		0 1	≡	
11		1	≡	0
12			≡	1 0

13			≡	false
14		false	≡	
15		false		

step	program	data	proc	proc-data
0	15 5 modulo 0 ≡			
1	5 modulo 0 ≡	15		
2	modulo 0 ≡	5 15		
3	0 ≡	5 15	modulo	
4	0 ≡	15	modulo	5
5	0 ≡		modulo	15 5
6	0 ≡		modulo	0
7	0 ≡	0	modulo	
8	0 ≡	0		
9	≡	0 0		
10		0 0	≡	
11		0	≡	0
12			≡	0 0

13			☐	<input type="checkbox" value="true"/>
14		<input type="checkbox" value="true"/>	☐	
15		<input type="checkbox" value="true"/>		

The programs above tell us that 10 is *not* a multiple of 3, and that 15 *is* a multiple of 5.

Puzzle 8.1

Now, it's your turn to use all the skills you have learned so far to build a program that solves the fizzbuzz puzzle. Follow the function design steps and break apart the recursive and base cases. When you get stuck, break out smaller parts of the program into named functions. Break the problem into smaller parts until you have something small enough you can understand and know that it works the way you want it to. Then, put the pieces back together.

Chapter 9

Programming a Car

I think people who write programs do have at least a glimmer of extra insight into the nature of God... because creating a program often means that you have to create a small universe.

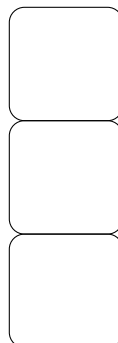
– Donald Knuth

Final Boss

This is the final boss level of the Programming Cards game. It is where we unleash our programming powers to defeat the most difficult puzzle. In this level we learn to program a car. We first learn how to interact with a toy car through programming, and then we create programs that make the car drive where we want it to go. This game can be as easy or as challenging as you want it to be.

The Road

The first part of the game is creating a road. You decide what you want the road to look like. To create the road, lay the cards face down but connected. To learn the game, we will have a simple road of 3 face-down cards that look like this.



The Car

We need a toy car. You can use a Hot Wheels toy or any mini toy car that will fit on one card. If you do not have a toy car, you can draw one on paper. You can even draw a car on one of the cards if you wish. It doesn't even need to be a car. It could be a pretty princess unicorn or whatever other toy or picture you choose. In this book we will call it a car, but use whatever you want. Now place the toy car on the bottom card closest to you, facing away from you.

Interacting With the Car

Forward

The process for the `forward` command is to move the car forward one step.

Function	Input	Output
<code>forward</code>	()	()

Rotate Right

The process for the `rotate-right` command is to rotate the car 90 degrees to the right.

Function	Input	Output
<code>rotate-right</code>	()	()

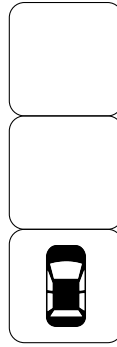
Rotate Left

The process for the `rotate-left` command is to rotate the car 90 degrees to the left.

Function	Input	Output
<code>rotate-left</code>	()	()

Programming the Car

Write a program that will drive the car down the road.



Required cards

cards	count
<code>forward</code>	2

step	program	data	proc
0	<code>forward</code> <code>forward</code>		
1	<code>forward</code>		<code>forward</code>
2	<code>forward</code>		<code>forward</code>
3	<code>forward</code>		

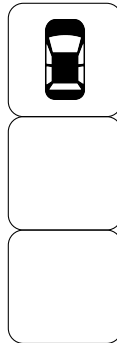
When the computer sees the `forward` card it knows to move the car forward one step. Now the road and the car should look like the following.



There is still a card on the **program** stack. It is the `forward` command. The computer executes it, and then everything should look like this.

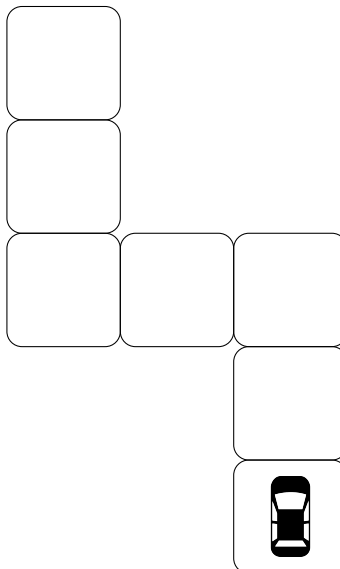
step	program	data	proc
------	---------	------	------

4			forward
5			forward
6			



Puzzle 9.1

Now, we can make more challenging roads. Write a program that will drive the car along this road.



Reading Sensors

In order to write programs that drive the car with more autonomy, we need to get feedback from the car. This is similar to getting input from the user. Right now, we can write programs when we know what the road looks like. We pre-programmed the turns. We want to be able to write a program for a road no matter how many turns or no matter where the turns occur. To do that, we need three more functions.

Scan Forward

The computer looks at what is directly in front of the car. If there is an open road square in front of the car, then the computer puts a `true` card on the **proc-data** stack.

Function	Input	Output
<code>scan-forward</code>	()	(Boolean)

Scan Right

The computer looks at what is directly to the right of the car. If there is an open road square to the right of the car, then the computer puts a `true` card on the **proc-data** stack.

Function	Input	Output
<code>scan-right</code>	()	(Boolean)

Scan Left

The computer looks at what is directly to the left of the car. If there is an open road square to the left of the car, then the computer puts a `true` card on the **proc-data** stack.

Function	Input	Output
<code>scan-left</code>	()	(Boolean)

Puzzle 9.2

Write a function that can drive the car down any road that has no forks. That means that you can build a road that has as many turns as you like but is a single lane. The program should be capable of getting the car from the start square to the stop square without knowing the road ahead of time.

Chapter 10

A Letter To Future Programmers

Programming is not a zero-sum game. Teaching something to a fellow programmer doesn't take it away from you. I'm happy to share what I can, because I'm in it for the love of programming.

– John Carmack

Build A Liberal Arts Foundation

Believe it or not, you are now a programmer. You know how to program if you understand how to craft instructions using conditionals and recursion. It may not be as glamorous as building an app or coding for a giant tech company, but the knowledge you gained is invaluable. By unlocking the fundamentals of programming you are now free to explore more advanced programming subjects. I believe the best way to increase your programming knowledge is to follow a liberal arts curriculum. In the information age your knowledge gives you value in the marketplace. Time is a finite resource and you must be selective about which knowledge you spend time acquiring. Much of my programming education could have been more efficient, and I spent years learning the ins and outs of specific tools and languages that I will never use again. As I give the following advice, I give it while imagining that I am talking to my past self. If I could do it all over again, what would I spend time learning?

Literature

As David McCullough said, "To write well is to think clearly." As programmers, we can learn a lot from good literature. The best authors are able to do something magical with language. Good nonfiction authors teach quickly and concisely. Good fiction authors inspire, show you new ways to view the world, and give insight into human nature. Don't overlook the power that the best books can have on your ability to program.

Music

To be a programmer you must also be a computer. You will need to juggle many things in your head at once and find patterns for reducing complicated chains of logic. Learning

how to play a musical instrument helps with both of these things. Develop the skill of sight reading and test yourself with complicated pieces. Learn music theory and you will discover how composers can turn something complicated into something simple and beautiful.

History

Whenever I think I have an original idea in programming, I search the past and find that someone else thought of it before me and did it better. If you are humble enough to learn from the past, you will grow significantly more than otherwise. However, studying history requires a critical eye for bias. There is always bias in history lessons. Even though a history book might be 100% factual, it is biased in its selection of facts. The authors are not necessarily trying to be nefarious, but this is the reality of studying history. Studying history is more than just learning facts, and it is more than believing what someone tells you happened. It is understanding the period in which people lived and why they made the choices they did. As you enter the programming world you will discover you are building on top of other programmers' work. I echo the sentiment of G. K. Chesterton when he said, "Don't ever take a fence down until you know the reason it was put up."

Language

Learning a second language is encouraged in a liberal arts education because it helps you understand your first language better. This also applies to understanding programming languages. As you learn a second language, you increase your critical thinking and communication skills. You will broaden your perspective on the world and expose yourself to different cultures and customs. Some languages are better than others at expressing certain ideas, and that is what you want to understand to be a good programmer.

Video Games

A good video game can have the same educational benefits as a good book and sometimes includes social interaction and teamwork. Just be sure to set appropriate limits for yourself. While there can be great educational benefits to playing video games, it is very easy to become a video game glutton and fail to live up to your full potential.

Hard Sciences

Some of the best programmers were, and are, video game programmers. Some of the most incredible programming feats happened because someone wanted a more realistic video game. Those programmers could accomplish what they did by leveraging their understanding of physics. Video games are just one example of how physics is helpful. As programmers our job is to model the world as it is; to do that, we must understand how the world works. Physics, Biology, Geology, and Astronomy enable you to program in various industries because they teach you how the world works.

Theology

A growing number of people believe that theology shouldn't be part of any curriculum. Still, the reality is that an education without theology is an incomplete education. Seek out answers to the questions: Who am I? Why am I here? Is there a God, and if so, what is man's relationship with God? If you believe you know the answers to these questions, don't shun this field of study. Instead, put your theory to the test against the greatest theologians. Contemplating these questions of the soul will give you perspective and direction and lead to a fulfilling life.

Economics

Even more pervasive than programming illiteracy is economic illiteracy. The brightest and most competent programmers fall prey to economic fallacies. Middle management and the C suite also neglect this field of study. Economics is the study of scarce resources. As programmers we must study how to utilize scarce resources like computing power or digital storage. As managers of programmers you must decide between the cost of building something with developer time or the cost of purchasing software outright. With a framework to reason about scarce resources you will make good decisions. You should pursue free market economics, known as the Austrian School of Economics. Discover the teachings of Thomas Sowell, Ludwig von Mises, Murray Rothbard, and many others, and you will understand the world we live in much more clearly.

Programming

A good programmer should understand many programming languages and master one or two. You want to learn from a wide range of types of languages and avoid learning many similar languages. Every time you learn a new language it becomes easier to understand the next. The fundamentals of these languages end up being very similar. The more significant differences come from how each language solves specific problems. There is something to learn from all of them.

Popular Programming Languages

Functional	System	Object Oriented	Scripting
Haskell	C	Java	Python
Elixir	Rust	C++	JavaScript
Racket		C#	Lua
Ocaml			Typescript

The Programming Cards game is a unique programming language specifically designed to help you see the parts that make up any programming language. Every programming

language is a pattern for building a list of instructions for a computer. Each language differs in how the list is constructed. In the Programming Cards game the list is a stack of cards, in other programming languages there are special symbols in a text file that are used to tell the computer when a command starts and when a command ends.

In the Programming Cards game, we discussed the importance of learning how to design functions because those are the building blocks of your program. You must develop that skill for the rest of your programming career. On top of that skill, you must understand the techniques used to compose functions. You learn this by learning algebra and abstract math, like category theory. However, even for veteran programmers, category theory is difficult to comprehend. One way to make the subject more accessible is to use a tool that helps you visualize and build an intuition for abstract concepts. I may write a sequel to the Programming Cards game to help teach this subject, but I recommend the Haskell language for now. Programming in Haskell will help build an intuition for the abstract ideas in category theory and give you practice for more advanced function composition.

Build a mastery of data structures. Many programmers stop their education on data structures after learning a handful of the most used ones. There are so many interesting ideas in data structure designs that will empower you as a programmer. As I said, I believe the list is the most important data structure. I said that because one can use the list as the building block for any other data structure. Don't just stop at learning the basics; aim to thoroughly understand the intricacies of each data structure, how they interconnect, and how they aid you in organizing your programs.

My last piece of advice is to work towards a larger project that interests you. Think of a problem you want to solve or a task you want to automate. Learning programming theory is essential, but it is also important to get practical hands-on experience. It's easy to fall into the trap of believing that you need to finish your education before you can start doing what you want to do in life. A liberal arts education is a lifelong pursuit; part of that pursuit is learning what you can do to improve the world. I hope you continue your programming education and build something amazing.

Appendix A

Solutions

Solution 1.1

program	
	1
	1
	1
	1
	1
	+
	+
	+
	+

Solution 2.1

program	
	3
	dup
	5
	>
	swap
	2
	≡

Solution 3.1

add10	
	10
	+

Solution 3.2

cube					
	<table><tr><td>dup</td></tr><tr><td>dup</td></tr><tr><td>*</td></tr><tr><td>*</td></tr></table>	dup	dup	*	*
dup					
dup					
*					
*					

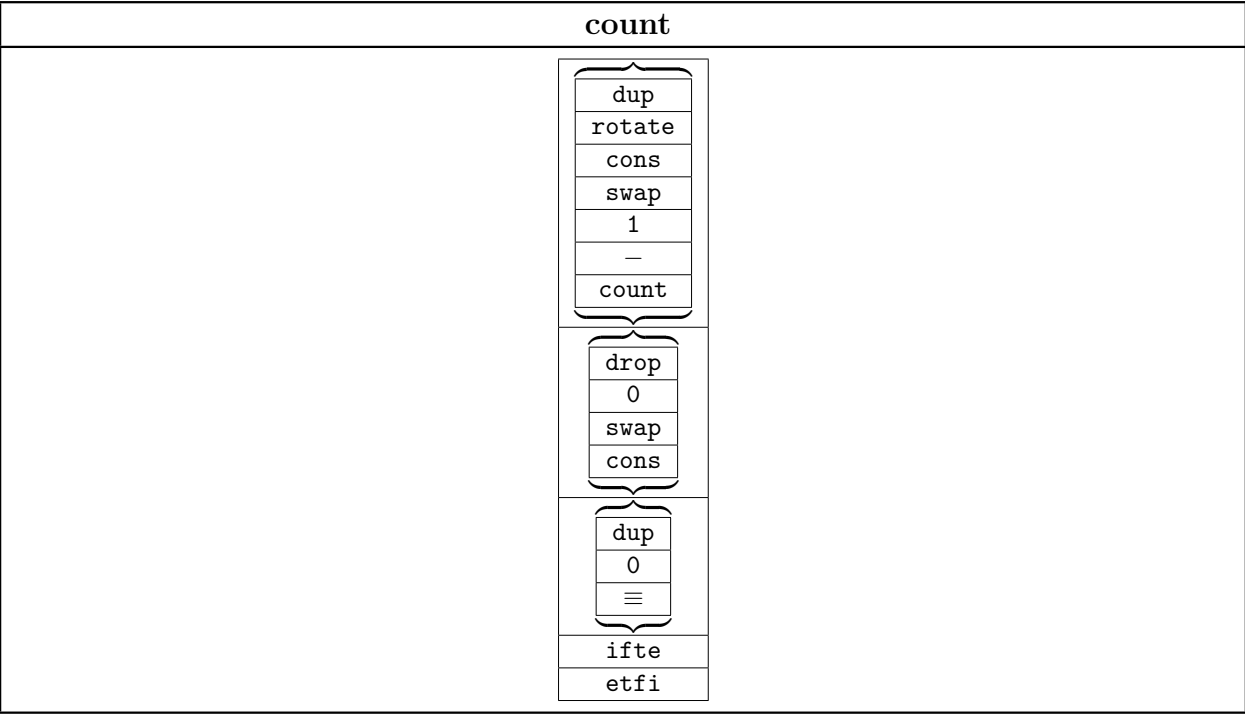
Solution 4.1

is-between?	in-or-out?															
<table><tr><td>dup</td></tr><tr><td>2</td></tr><tr><td>$\geq$</td></tr><tr><td>swap</td></tr><tr><td>4</td></tr><tr><td>$\leq$</td></tr><tr><td>$\wedge$</td></tr></table>	dup	2	$\geq$	swap	4	$\leq$	$\wedge$	<table><tr><td><table><tr><td>"outside"</td></tr></table></td></tr><tr><td><table><tr><td>"inside"</td></tr></table></td></tr><tr><td><table><tr><td>isBetween?</td></tr></table></td></tr><tr><td>ifte</td></tr><tr><td>etfi</td></tr></table>	<table><tr><td>"outside"</td></tr></table>	"outside"	<table><tr><td>"inside"</td></tr></table>	"inside"	<table><tr><td>isBetween?</td></tr></table>	isBetween?	ifte	etfi
dup																
2																
$\geq$																
swap																
4																
$\leq$																
$\wedge$																
<table><tr><td>"outside"</td></tr></table>	"outside"															
"outside"																
<table><tr><td>"inside"</td></tr></table>	"inside"															
"inside"																
<table><tr><td>isBetween?</td></tr></table>	isBetween?															
isBetween?																
ifte																
etfi																

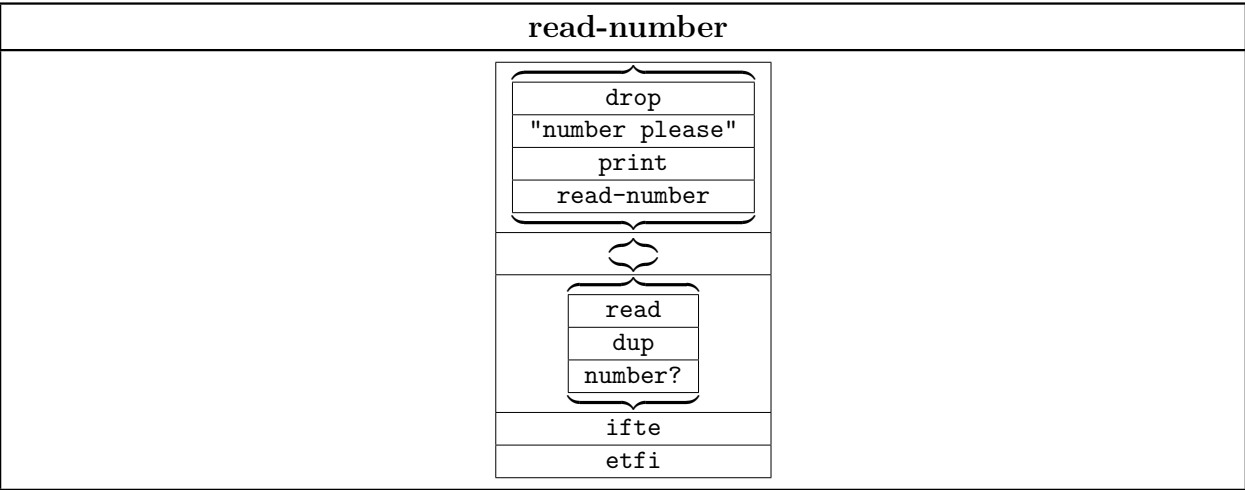
Solution 5.1

countdown					
<table><tr><td> dup 1 — countdown </td></tr><tr><td> </td></tr><tr><td> dup 0 $\equiv$ </td></tr><tr><td>ifte</td></tr><tr><td>etfi</td></tr></table>	dup 1 — countdown		dup 0 $\equiv$	ifte	etfi
dup 1 — countdown					
dup 0 $\equiv$					
ifte					
etfi					

Solution 6.1



Solution 7.1



Solution 7.2

add-user-input					
	<table><tr><td>read-number</td></tr><tr><td>read-number</td></tr><tr><td>+</td></tr><tr><td>print</td></tr></table>	read-number	read-number	+	print
read-number					
read-number					
+					
print					

Solution 8.1

multiple?	mult3?	mult5?	mult3-and-5?												
<table><tr><td>modulo</td></tr><tr><td>0</td></tr><tr><td>≡</td></tr></table>	modulo	0	≡	<table><tr><td>3</td></tr><tr><td>multiple?</td></tr></table>	3	multiple?	<table><tr><td>5</td></tr><tr><td>multiple?</td></tr></table>	5	multiple?	<table><tr><td>dup</td></tr><tr><td>mult3?</td></tr><tr><td>swap</td></tr><tr><td>mult5?</td></tr><tr><td>∧</td></tr></table>	dup	mult3?	swap	mult5?	∧
modulo															
0															
≡															
3															
multiple?															
5															
multiple?															
dup															
mult3?															
swap															
mult5?															
∧															

show	show-fizz-or-buzz	show-buzz-or-num
dup show-fizz-or-buzz "fizzbuzz" print dup mult3-and-5? ifte etfi	show-buzz-or-num drop "fizz" print dup mult3? ifte etfi	print drop "buzz" print dup mult5? ifte etfi

fizzbuzz
show 1 + fizzbuzz drop dup 101 ≡ ifte etfi

Solution 9.1

program
forward forward rotate-left forward forward rotate-right forward forward

Solution 9.2

drive	check-left	check-right
check-left forward drive scan-forward ifte etfi	check-right rotate-left forward drive scan-left ifte etfi	rotate-right forward drive scan-right ifte etfi

program
drive

<https://programmingcardsgame.com>

