

+ cover image

bookish.press/book/chapter1

Chapter 1

Introduction

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

What motivated me to design this book? There are already many great books on programming languages. Existing books on programming languages can teach you how to use advanced programming language features, implement languages, reason about the mathematical theories underlying languages, reason rigorously about correctness, and so much more. Experts in human-centered computing have written countless books of their own, which can cover that topic in far greater depth than a book that approaches human-centered computing through the lens of programming languages. Why, then, does the world need a book that puts these two perspectives together?

The answer is that this book was created to address the core challenge I face in teaching the programming languages electives at WPI: modern students deserve a course and a book which address programming languages as a breadth topic. My students will go on to a wide range of careers: many will be professional programmers, some will transition into a job where their computer science knowledge takes on a background role as it informs their knowledge of the world around them, and some will continue down the academic road. I want this book to provide something for all of them, whether graduate or undergraduate, computer science major or not. The breadth approach accomplishes this. Breadth is also best-suited for the realities of the digital age. If an alum forgets the details of a classic algorithm five years after graduation, they can always look it up, but if a student graduates without being shown the breadth of the discipline, they might never think to go looking for what they need.

At the same time, this book will not be devoid of classic algorithms and classic results; a breadth approach includes these, but does so in moderation. In fact, we will start narrowly with the classics, teaching core technologies used for language implementation and core vocabulary for technical discussion of languages, then branch out into an increasingly humanistic approach, incorporating human-centered works which do everything from

perform user studies to guide language iteration to employ humanistic critiques to consider the role of programming languages in contemporary social issues. We will have a little fun along the way, using programming languages for creative expression as case studies in how the disciplinary and interdisciplinary sides of mind can collaborate in the work of programming language design. At times, this book will depart from the norms of the textbook genre by incorporating small novel research results or **autoethnographic** experiences from the author's life. When we do so, the discussion will remain scholarly. Learning how to explore previously-unexplored ideas and how to bring one's own experience into a rigorous academic experience are both part of a complete education.

This book is designed in the paradigm of open education, meaning the author publishes the full teaching materials openly so that self-taught students can engage on their own and so that educators at other institutions can easily incorporate this work into their own. In addition to the typical exercises presented in textbooks, this book includes recommendations for more comprehensive assignments or projects, as well as direct suggestions for activities that teachers can perform in a classroom. These recommendations serve a dual purpose to the author's students at WPI, to help them prepare for an active class session and get the most out of it. Only solutions to WPI

homework assignments are left unpublished, and instructors elsewhere are encouraged to contact the author for access. Though solutions are unpublished, autograding code for assignments is intentionally published because this approach is closest to real-world practice. Real-world programmers often debug their code by exploring its behavior on a variety of test cases, so this skill is valuable use of a student's time, but guessing at the grader's inner workings or intentions would distract from the heart of the educational experience.

For students in the accompanying course: in keeping with this spirit, we will engage with, but not fully commit to, an educational philosophy called **ungrading**, a philosophy that says traditional grades are counterproductive in that they replace a student's natural intrinsic motivation for the submit with the external motivation to optimize for high grades. The exact grading structures will vary from year to year and students should always consult the latest syllabus, but in general I seek to strike a balance between students' desire for me to provide them with a source of external accountability and my desire to provide space for free exploration of aspects that interest you. On that note, the references at the end of each chapter are not merely good scholarly practice, but they are an opportunity for you to dig further in the topics that most interest you and to engage with the latest research on the subject.

When I tell students about programming languages, their first question is often “which language?” On one hand, the differences between different programming languages do matter in a significant way, otherwise we would not study these differences. On the other hand, there is no one right language, not only because different languages are best-suited to different needs of different programmers, but because there is significant cross-pollination between modern languages, and the most important programming language features often reoccur across many modern languages. Different languages are used in different courses or even different iterations of the same course for logistical reasons, so the majority of this book is written in a language-neutral way. As separate chapters, I include primers on several languages that are appropriate for use in a programming languages course. Though there is no perfect language, it is important to provide background material for some language, to support you in implementing the programming language technologies discussed throughout the chapters on language implementation. The exercises attached to those chapters will encourage you to record your experiences with the given language and to reflect and interpret those experiences.

The human-centered nature of this book is fundamental to its character, and the choice to be human-centered is unapologetic, because this decision can help students understand how to apply

the material in their daily working life, help a broader range of students see themselves in their work, help students access the most up-to-date research in the field, and even help prepare students for future research careers. At the same time, I am aware that some students are not used to engaging with their own personal identities in a computer science classroom, nor the potential for political discourse that comes with discussion of social issues in the classroom. We will engage with potentially-personal topics, but will do so in a professional and academic way, where what matters is whether you demonstrate mastery of the works we study, not your personal belief.

Classroom activities:

1. I recommend that instructors start the first day of class by sharing a written classroom climate statement, which is often part of the syllabus. This statement should come from the heart and should not be boilerplate. My own statement usually speaks from my positionality as a disabled trans woman to remind students that they belong in my classroom, establishes mental health as a priority, and sets the expectation that students are to go out of their way to make one another feel welcome in the classroom, and that students should question me openly while showing respect for my expertise. In the discussion of mental health, I inform students of mental health resources contained in the syllabus.

2. Make time on the first day to go over relevant syllabus information. Office hours and other supports should be explained in a way that emphasizes students should seek help whenever that help could be useful, not only when they have a narrow specific question or when they have exhausted every possible option. Waitlist status should be discussed, as should software and tooling questions.
3. The first day should include social interaction. I have student introduce themselves I encourage but do not require students to share pronouns (so as not to put pressure on closeted students). I will start group discussions by asking students to share (i.e. vent) past experiences with and current opinions on different programming languages.
4. To the extent possible, leave time to install relevant software in class, and provide technical support for the installation process.

Exercises

1. Review the following list of learning objectives for the book: Identify problems where programming language design can be used, Communicate with clarity and technical depth about language design, Develop a mathematically-precise definition of a language's syntax, Develop a mathematically-precise definition of a language's semantics, Implement interpreters for programming languages, and Situate your own work among the schools of thought discussed in class. Which of

these objectives interest you the most? What objectives of your own do you have, which are not on this list? Skim through the chapters of the book and identify any material in the book which might help support your objectives. Share your objectives and interests with the course staff.

2. Maintain a list of questions you have about programming languages in your daily life. As you continue through the book, try to elaborate these questions into a style that can be solved rigorously, such as falsifiable questions that could be studied experimentally or subjective questions which could be justified from reasoning from well-stated assumptions.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter2

Chapter 2

What is a Language?

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * We will explore what the term “Programming Languages” should mean and why an all-encompassing definition is challenging to find
- * We will frame the scope of the book through competing schools of thought representing the different humans on which on which the study of programming languages is centered
- * We will compare this approach to an alternate organizing principle for the study of languages - organization around programming language paradigms

As we undertake the study of programming languages, it is only fair that we begin with defining our object of study. We ask ourselves earnestly: “What is a Programming Language (PL)?” At first glance, this question may seem simple to answer, but the closer we look, the harder it becomes to arrive at a universal definition. If you are reading this book, you can most likely name some examples of programming languages: procedural languages like C, object-oriented languages like Java, or dynamically-typed languages like Python. If you have branched out and explored the diversity of programming languages, you might recognize functional programming languages like Lisp or Haskell. Each of these is a valid example of a programming language, and you would be correct to call them as such. However, examples alone are not enough to define the category of programming languages, because they do not reflect the full breadth of the topic, especially when we wish to approach the study of programming languages in an interdisciplinary fashion. Instead, we will work toward a definition of a programming language in an iterative way, proposing definitions and then challenging them. We begin with a simple definition.

Potential Definition: A programming language is a formal notation for specifying instructions executed on the central processing unit (CPU) of a computer.

Challenge: We have already encountered an example of a language that does not meet this definition: Python. Python programs are most often executed in an *interpreter*, i.e., a program which runs other programs. As such, it would be inaccurate to define the meaning of a Python program by defining a set of CPU instructions, because Python does not define itself this way, but rather defines itself primarily via interpretation. Though there exist compilers which transform Python programs into CPU instructions, they are not the primary route through which Python programs are understood. The closer we look at this definition, the more it falls apart. Though Java is compiled in typical usage, it is not compiled to CPU instructions, but to an abstract low-level language called the Java Virtual Machine (JVM). Java is not typically interpreted, yet the definition does not apply. Even C, which is most often compiled, does not meet this definition because the language standard does not specify the exact instructions corresponding to each program, nor could it, because all three of these languages are intended to be platform-independent and to support execution on a wide array of CPU types. Even notwithstanding the diversity of CPUs, this definition emphasizes the implementation choice between an interpreter (Python), traditional compiler (C), or bytecode compiler (Java), which belies the reality that a single programming language often has multiple implementations which employ competing approaches: there are interpreters for C and there are compilers

for Python. Confronted with this contradiction, we will attempt to redefine programming languages using an abstract model of computation, which is machine-independent. There are many such equivalent models, such as Turing machines and lambda-calculi; we define programming languages in terms of Turing machines because they are widely-taught.

Potential Definition: A programming language is a formal notation wherein every notationally-valid string of text corresponds to some Turing machine.

Challenge: Turing machines formalize a limited notion of computation. Turing machines can account for input-output behavior (which input string corresponds to which output string) and termination (whether or not a program runs forever), but even the core concept of program state is addressed in a limited way, using a tape, which is a modeling abstraction that rarely corresponds to real programs nor real computers in the modern era. This discrepancy means that other core concepts like execution time (how long a program takes to compute its output) cannot be modeled in a way that reflects modern reality. Other aspects of program behavior such as communication between concurrent programs, back-and-forth interactions with a user, and power usage are completely unaccounted-for. The more broadly we cast our net, the more thoroughly the Turing machine metaphor breaks down. If we wish to study at all how a user

engages with the notation of a programming language, e.g., which aspects they find usable and which aspects they find confusing, then a computational model such as a Turing machine is of little use. We now revise our definition to account for this challenge by refusing to commit to a specific model of computation.

Potential Definition: A programming language is a formal notation wherein every notationally-valid string of text corresponds to some computation.

Challenge: Though this definition has improved to the point where it encompasses all our example languages thus far (C, Java, Python, Lisp, Haskell), it does not capture the discipline of Programming Languages as a whole, nor does it capture the subject matter of this book. We challenge the definition by presenting a wide range of objects-of-study. The skills we study when we study programming languages are applicable to all the following things, so it stands to reason that a definition of programming languages should encompass them:

- * Declarative programming languages are programming languages that operate by defining a problem, rather than defining its solution. As such, they do not define computations, per se:
- * Logic programming languages such as Prolog define sets of rules, from which computations can be performed, but

which do not describe a computation directly.

- * Markup languages such as HTML (Hyper-Text Markup Language) or XML (eXtensible Markup Language) define data, but not computation. They are often excluded from definitions of programming languages because they are not programmatic, but are included in our definition of a programming language because they share a common structure with programmatic languages.
- * Domain-specific languages (DSLs) are programming languages optimized for a certain application domain. Many DSLs meet the definition of programming language as a computational language, but others do not.
 - * The GL Shader Language (GLSL) and other shader languages are used to program shaders, specialized programs used for computing on a Graphical Processing Unit (GPU). Shader programs originate in graphically-intensive applications such as video games and digital animation, but have found use in diverse parallel computing tasks including scientific computing and machine learning.
 - * Permission policy languages are used to let users define complex policies for managing their own security and privacy.
 - * Macro languages (e.g. AutoHotKey) are used by end-users to improve productivity in diverse end-user computing tasks.

- * The term “little languages” denotes particularly specialized DSLs. In contrast to other DSLs, the development of a little language is usually emergent from the development of a larger software project, where the need for a programming language, or something like one, is discovered over time.
 - * Many software applications employ custom textual languages for *configuration files*. These languages are potentially as complex as the configuration of the program itself.
 - * It is a common practice to maintain plain-text *logs* of events that occur during the lifetime of an application. *Log formats* are little languages in their own right, and logging applications may contain *query languages*, little languages used to search through log data.
 - * Beyond log languages, plain-text and binary data formats used by any application can be understood as little languages.
 - * Pseudo-languages are programs which are *not* programming languages, but can be studied using the same approaches we use to study programming languages, because they work in a similar way to programming languages.
 - * Spreadsheet software such as Microsoft Excel and Apple Numbers are similar to programming languages because spreadsheet formulas express computations. By

understanding spreadsheets through the lens of programming language semantics, one can elucidate how formula computations work and why; see *Self-Adjusting Computation* for a programming foundation with similar dynamics to a spreadsheet. Conversely, spreadsheets have something to teach programming language scholars: spreadsheets are used successfully on a daily basis by a large number of users who have never been trained in programming, and thus might provide insights on how to make programming accessible to broad audiences.

- * Email filtering rules, such as “if the sender’s address contains ‘wpi.edu’, then move the email to the spam folder” have a conditional “if-then” structure, familiar to many programming languages. Beyond this superficial similarity, the “if” conditions of an email filter constitute a form of pattern-matching, a programming language feature that allows programmers to express rich conditional branching based on the structure of data values. Research on pattern-matching, such as research on efficient compilation techniques, could shed light on how to efficiently process large sets of email filters.
- * Effects in media (i.e., image, audio, and video) editing programs have the structure of a program. The audiovisual media are the inputs and outputs of the program, and each effect is an operation within the program. Examples of effects are scaling, rotating, and

translating images, slowing down or speeding up a video, or applying a compressor to reduce the dynamic range of an audio track. Programming language techniques could be used to reduce the incidence of errors in video effects that might take hours to render, simplify maintenance of complex effects, or optimize execution.

Through this lens, it becomes apparent that programming languages are all around us. To care about programming languages, you do not need to design new programming languages, do mathematical proofs about language semantics, or implement new compilers and interpreters for a living. At some point in their career, most programmers will not do those things, but *will* write little languages or pseudo languages, because most major software applications have data and configuration, and many provide users to give rich, structured, computational inputs. For that matter, you do not even have to be a programmer, because everyone from office workers to content creators engages with software that reflect the ideas we encounter in the study of programming languages. Once you look at PL this way, you can see why PL is for everyone. Therefore, this book chooses to define the field of programming languages by looking at the people who study them:

Our Definition: A programming language is anything that is studied by people who “study programming languages”.

Challenge: This definition is frustratingly circular. It tells you nothing about what this book studies. This frustration is intentional, however. Different people think about programming languages in such profoundly different ways that it could be difficult to converse with one another if we do not take this moment to present the diversity of approaches with which languages are studied. Throughout this book, we frame the diversity of the field by dividing the study of programming languages into different “schools of thought;” these categories are not meant to be rigid, but to loosely define how different academic disciplines and groups outside academia engage with the field. The title of this book, “Human-Centered Programming Languages” does not simply mean that we look at the humans who use programming languages as programmers, but that we look at the humans who study languages and their breadth of motivations, both human-centered motivations and others.

Schools of Thought

This book emphasizes competing schools of thought because, in so doing, we can explore the different modes of thinking and ways of knowing that appear in different communities, thus enriching our own understanding of the field. This book is not here to tell you which school of thought you should belong to. It’s here to provide you breadth so that you may decide for yourself where you would

like to situate yourself in the field. In this book, we explore the following schools of thought: “Practitioner,” “Implementer,” “Software Engineer,” “Social Scientist,” “Humanist,” and “Theorist”. We summarize each school of thought in turn. Each school of thought is accompanied by a cartoon which will appear throughout the book when we practice the given school of thought. These cartoons are modeled on real people who engage in the given school of thought.

The Practitioner

This is the biggest school of thought: people who code. If you are in this school of thought, you are the largest audience for this book! To characterize the perspectives of the Practitioner, one can look at social spaces where programmers gather, such as StackOverflow, GitHub, and social media. Practitioners are often goal-directed: they have specific programming tasks they wish to complete as part of their job or hobby, and their interest in programming languages relates to helping them complete their specific goals. Just because the Practitioner is goal-directed, does not mean they are any less passionate. Practitioners often have strongly-held beliefs about programming languages which are drawn from their extensive lived experience. They might extensively discuss user-facing aspects of languages such as their surface syntax, error messages, compile times, tooling, and

standard libraries. Because of the focus on user-facing behavior, their explanations of their positions may be particularly accessible to other programmers. On the flip-side, the personal focus is sometimes accompanied by interpersonal hostility in social spaces, with negative implications for inclusivity. In this book, we seek to respect the lived experience of practitioners while keeping discourse to a professional, impersonal, and all-inclusive level.

Examples of books the Practitioner might read are “The Practice of Programming, The Pragmatic Programmer” and programming language-specific books like “The C++ Programming Language” or “The Rust Book.” The Practitioner’s fundamental question is “How do I write this program?”

The Implementer

An Implementer is anyone who implements a programming language, typically as a compiler or interpreter. By definition, they are simultaneously a Practitioner, but they have an additional set of needs and concerns. The Implementer is heavily concerned with tools for building languages. The implementer might use regular expressions, context-free grammars, and parsing expression grammars to implement a parser and abstract syntax trees to represent parsed programs. In a compiler, intermediate

representations such as the Low-Level Virtual Machine (LLVM) may be used for optimization and code generation, while an interpreter may simply recurse over the structure of a program to compute a direct result. The Implementer needs a strong understanding of a program's meaning (semantics) insofar as they need to faithfully capture a program's meaning in the implementation.

Examples of books the Implementer might read are “Introduction to Compilers and Language Design” by Douglas Thain or “Compiler Construction” by William M. Waite and Gerhard Goos. The Implementer’s fundamental question is “How do I implement this programming language?”

The Software Engineer

The Software Engineer is the first of our human-focused schools of thought. Software Engineering concerns itself with a specific set of humans, typically professional programmers working in large teams or organizations. Of all the schools of thought, Software Engineers are the most concerned with scale, collaboration, management, and process. Because Software Engineers target professionals as their object of study, they have a flexible range of solutions at their disposal. When met with a problem in software development, the solution could lie with software (new languages,

new tools), with people (new training, management processes, or quality assurance processes), or both (training developers to use a new tool or language). Both the language and person can change. Therefore, the Software Engineer school cares to study how professional programmers use programming languages and to design languages and language tools based on how professional programmers are likely to use them, but does not treat current practice as fixed and unchangeable.

Examples of books the Software Engineer might read are “The Clean Coder” by Robert C. Martin, “Essentials of Software Engineering” by Frank Tsui, Orlando Karam, and Barbara Bernal, or “The Mythical Man-Month” by FP Brooks Jr. The Software Engineer’s fundamental question regarding programming languages is “How can professional software developers, programming languages, and software development tools best work together to create quality software?”

Model?: Michael Coblenz (he/him) is faculty in Computer Science & Engineering at University of California San Diego.

The Social Scientist

Social Science overlaps significantly with Software Engineering, but is a much broader term. The Social Scientist’s emphasis is on the

rigorous academic study of humans. Compared to the Software Engineer, a Social Scientist may place less focus on immediate technical solutions; whereas a Software Engineers might employ social science methods to make better languages or tools, the Social Scientist may also study humans for humans' sakes. They often study social issues within communities of programmers and computer users. Who do these communities include or exclude? Why? What could be done about that? These questions are answered using scientific approaches, which can encompass both quantitative or qualitative approaches.

Examples of books the Social Scientist might read include "Working in Public: The Making and Maintenance of Open Source Software" by Nadia Eghbalee. The Social Scientist's fundamental question about programming languages is "How can the impacts of programming languages on communities of people be measured?"

The Humanist

This school also studies social issues. It asks who communities include or exclude? Why? What could be done about that? The difference is that it uses methods from the humanities. Important books about computing could be read closely and their language analyzed. People look at rhetoric about languages and rhetorical

structure present in code itself. People do theory-building, taking core ideas from social theorists and applying them to the specifics of PL communities. This school is relatively small, but exciting.

Examples of books the Humanist might read are “Persuasive Games: The Expressive Power of Videogames” by Ian Bogost and “Rhetorical Code Studies” by Kevin Brock. A fundamental question for the Humanist about programming languages is “How can our understanding of how social structures operate and our understandings how programming languages operate inform one another?”

Model: Yunus Telliel (he/him) is an anthropologist at Worcester Polytechnic Institute and an expert in computing ethics. He has collaborated with the author on a rhetorical approach to programming language design.

The Theorist

This school says that PLs are formal languages that can be defined and analyzed mathematically. A “good PL” is a language that we can analyze in powerful ways. A Type Theorist is a Theorist who believes a “good PL” has a rich static type system that lets us prove a soundness theorem, which is something like:

Theorem [Type Safety]: “Any time your compiler accepts a program from you without giving an error message, that program will satisfy *some specific notion of correctness, chosen by the designers,*” which may include:

- * If program has type t, then the result has type t
- * If program has a type, then no segfaults can occur
- * No data races nor deadlocks occur
- * No memory leaks occur

The Theorist understands programming language design as the task of abstraction-building, the discussion of programming languages as abstraction-criticizing, and programming as the use of abstractions.

Examples of Abstractions:

- * Types: “I don’t care what x is, as long as it’s an integer”
- * Semantics: “I don’t care about compiler implementation details”
- * Correctness: “I don’t care what program you write, as long as it does X”
- * In C: divide program into procedures (like functions, but have state)

- * In Java: divide program into objects. Objects combine state and code, but hide some of it, making it abstract
- * In Racket: divide main function into helper functions. In Racket/Lisp: all language features are nested lists with parentheses. This enables *homoiconicity*, representing programs using the language's core datatypes.

Theorists may appreciate abstraction-building as a mental workout that combines rigor and creativity. However, there are important and severe limitations if this is the only way we ever think. These limitations have been studied in the humanities for decades in the contexts of philosophy and logic, which are intimately connected to PLs. The risk of abstraction-building is that it is reductionist. In the humanities, this poses great dangers; we all know what happens when we reduce a person to a narrow aspect of their identity and declare them to be the Other, or when the people in power choose which aspects of your identity you're allowed to care about and how. Reductionist risks do transfer to programming languages world. Theory culture could be rightly criticized that it not merely emphasizes mathematical rigor but often mathematical competition in a fashion that risks telling newcomers that they don't belong. They have also been rightly criticized for de-emphasizing practice and application so much that they end up significantly narrowing their audience.

Examples of books the Theorist might read are “Types and Programming Languages” by Benjamin Pierce and Concrete Semantics by Gerwin Klein and Tobias Nipkow.

It’s important to know that these different schools of thought exist, because it allows you to recognize that each of these groups uses different vocabulary to talk about programming languages, uses them in different ways, and has different conversations. A speaker who uses vocabulary from the Theory school, might struggle to communicate with an audience which primarily uses those words as they are used in the Practitioner school. If each school has its own language or dialect of speaking, the goal of this book is to make the reader bilingual or even a polyglot.

Programming Language Tourism

The study of programming languages can often be organized as a tour. This book is a tour of *the different schools of thought*. Other ways to study programming languages include touring a single school of study in depth, touring programming languages, or touring programming language paradigms. We discuss how programming languages could be studied through the lens of paradigms, mainly to justify why this book is not organized in such a way.

When you search for basic information about a programming language, it often comes with a list of language paradigms, such as:

- * Imperative
- * Functional
- * Logical
- * Object-oriented

These are meaningful technical terms that can be given meaningful definitions:

- * “Imperative:” The programmer’s core mental abstraction is program state
- * “Functional:” The programmer’s core mental abstraction is the input-output behavior of mathematical functions
- * “Logical:” The programmer’s core mental abstraction is logical proof
- * “Object-oriented:” The programmer’s core mental abstraction is objects

Lists of paradigms often include other terms like “scripting language”, “high level language”, or “declarative language”. These terms do not have clear technical definitions, but they have cultural significance to Practitioners who use them: a “scripting language” may describe a language which is often used for short one-off programs that do not require maintenance, a “high level language” may describe a language where the Practitioner experienced little need to think of system-level implementation details, and a “declarative language” may describe a language where the Practitioner experienced a focus on defining problems over defining computations. If these paradigms are meaningful, why are they not the basis for organizing this book?

1. In the modern age, learning about paradigms no longer requires a guided teaching experience. Programmers can learn about paradigms throughout their career by self-study or peer-learning.
2. Modern programming languages mix these paradigms freely. As programmers, our abilities of self-expression have reached the level where we do not need these boxes anymore.
3. This book seeks to provide lasting, generalizable knowledge, yet help you connect that knowledge to what happens in your real life and career. The “Schools of Thought” approach gives you a broader mental map of generalizable knowledge.

Classroom Activities

- * Have the students work together to reach a working definition of what a programming language is.
- * Discuss: When programming, what languages do you consider using, why, and what weaknesses come with that choice?

Exercises

- * Which schools of thought are most likely to influence your project, and how? Any school of thought that you're specifically not interested in?
- * Go to a website where people discuss programming languages. Record some of the questions they ask each other. Which schools of thought could address these questions, and how?
- * Write a list of programming abstractions you have used before.
- * Little-language hunt: Take a software application you use frequently and explore its functionalities, trying to identify little languages and pseudo-languages within it.

- * Research project: Typed Video Editing. Propose a new design for a video editing effect system which satisfies a Type Safety theorem where all valid effects can be rendered without error, e.g., without missing data or invalid arguments.
- * Research project: Design and implement a term project. This book is suitable for use in a project-based curriculum. The following section provides advice on how a “build-your-own-language” project could be structured with an emphasis on the Implementer and Theorist. From the perspective of the Implementer, the key parts of a language are its syntax and meaning (dynamic semantics), in addition to which the Theorist is also concerned with type systems and correctness.
 - * Syntax (Is this a program? Which one?) For the first stage of a term project, specify the syntax of your programming language. Use regular expressions to describe the basic building blocks of a program and use context-free grammars to describe how the pieces fit together.
 - * Execution (How do I run it? What happens then?) Study the chapter on operational semantics to learn how to give precise specifications of program behavior. Even for coding-heavy projects, it’s great to outline the semantics first, because it serves as a guide for your implementation. Programming languages can be implemented with interpreters (another program, that runs it) or compilers (a program that transforms it into another language, which could be either high-level or

low-level). You are welcome to do either, but this book emphasizes interpreters over compilers for reasons of scope.

- * Types (static semantics) Type Theorists care **a lot** about “What can I learn about your program, without running it?” because this question is closely related to “What can a programming language designer do to put more correct code in the world?”. The first step is typically to develop a type system. This is a set of mathematical rules that assign types to programs and, importantly, tell you if a program is well-typed (has a type) vs. ill-typed (does not have a type at all).
- * Correctness (Soundness+Verification) State and prove a correctness theorem such as Type Soundness: every well-typed program runs without certain kinds of errors, and if it terminates, it gives back an answer of the right type.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter22

Chapter 3

Programming in Rust

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

This chapter serves as a primer for the programming language Rust. It assumes that the reader has prior experience in several different programming languages, but no prior experience in Rust. The intent is to provide enough knowledge to perform coursework in Rust while simultaneously introducing core vocabulary for the discussion of programming languages.

Why Rust

Every programming language has a reason it was created, a selling point promoted by its advocates. The reason Rust exists is to overcome the tradeoff between speed and safety. We discuss that tradeoff in order to explain the design of Rust.

Speed vs. Safety

The programming languages best-known for their speed are simultaneously known to be unsafe, particularly the languages C and C++. Why are C and C++ unsafe, why are they fast, and how does a language achieve speed without sacrificing safety? When we say C and C++ are unsafe, we specifically mean they are memory-unsafe, which means that it is possible for data values in memory to be corrupted, used after resources are freed, and double-freed, and that it is possible for a program to crash with a segmentation fault, which is an attempt to access data at an invalid memory location. Memory-unsafe also goes hand-in-hand with the potential for memory leaks, bugs where resources are not freed after use, typically causing long-running programs to exhaust their resources and ultimately crash. Memory safety has a clear cause: the reliance on manual schemes for memory allocation, where the programming calls a function malloc (resp. the new operator in C++) to allocate new memory and free (resp.

the delete operator in C++) to deallocate memory after it is no longer needed. These classes of errors are simply impossible in languages that use a technique called garbage collection to provide automated memory management; garbage-collected languages are memory-safe.

The relationship between garbage collection and speed has historically been a complicated one. Early garbage collectors were slow, but throughput improved over time to the extent that extreme cases could be engineered in which the amortized throughput (i.e., the average performance across a large number of operations) of garbage collectors exceeded the performance of manual allocators. As throughout increased, latency often remained an issue: otherwise efficient collectors may exhibit rare but long pauses in application code as collection is performed, posing a usability problem in software containing graphical user interfaces and a safety issue in control software for physical systems. Over the years, garbage collection techniques with latency guarantees emerged, amelioriating this problem. Though garbage-collected languages do not have a monopoly in the contemporary programming landscape, their usage is far more widespread than in past decades, and many application-level programmers never encounter a programming task where garbage collection poses an insurmountable performance barrier.

Rust is a recent invention compared to both C and garbage-collected languages, and its invention is motivated by the view that performance will always be driving concern in systems-level programming. Systems-level programming serves to create infrastructure on which third-party code will execute, such as an operating system or web browser rendering engine, the latter of which was the first major successful application of Rust. Because there is no limit to the potential complexity of future third-party code, it is a priority to provide high performance and minimize barriers imposed on code written by others. In order to distinguish itself from both the thoroughly established usage of C and C++ and the high-quality garbage collectors available in application-level programming languages, Rust needed a distinct approach: use a static (i.e., compile-time) type system to rule out memory safety errors instead of garbage collection. This line of thinking fits neatly into the Theorist's worldview: an unending array of programming language research projects have used advanced static type systems to guarantee that complex correctness properties will hold for programs at runtime. Where Rust has distinguished itself however, is by breaking out of the research sphere into the applied sphere, with robust community, libraries, and tooling.

Theory and Practice

Rust makes an interesting case study for the transfer of programming language theory to practice because its type system has a clear theoretical basis: substructural type systems.

Substructural type systems are type systems which give up one or all of the following so-called structural principles:

- * Contraction: Variables can be duplicated freely
- * Weakening: Unused variables are acceptable
- * Exchange: The order in which variables are defined does not limit the order in which they are used, within their scope

Every potential combination of these rules leads to a distinct type system. Linear types, which permit Exchange but neither Contraction nor Weakening, have long been used to model resource (e.g. memory) management. Rust uses an affine type system, which permits both Exchange and Weakening but not Contraction. Compared to linear types, affine types only add a modest assumption: affine types assume that all data values are capable of being deleted.

By removing the Contraction principle, affine types provide a major advantage: variables correspond directly to the ownership of memory, thus it is easy to automate memory management

without relying on garbage collection: resources are freed when the corresponding variable goes out of scope. No advantage comes for free, however: affine typing means that shared references to the same piece of data are significantly more complex and that the need to copy data manually is increased. Nor is the payoff for affine typing restricted to management: the reliance on copying over sharing also greatly simplifies the development of correct parallel programs, i.e., programs which run across multiple processors at the same time.

The transition of theory to practice presented by Rust is potential interest to any reader of this book, but it will ring a particular bell for students at WPI, reflecting the school motto “Lehr und Kunst” (roughly “Theory and Practice” in German).

Community

Though overcoming the speed-safety tradeoff is a major motivation for the use of Rust, it is not the sole motivation. As of this writing, Rust has grown quickly in popularity over recent years, which has been cited in the research literature as a motivation for increasing the availability of Rust educational materials. The fact that Rust has achieved increasing popularity while incorporating advanced type system features sends a valuable message in and of itself, that the theory of programming

languages can create real-world impact on the practice of programming.

Rust is also chosen for its inclusion in this book because of its community. The Rust community has had no shortage of its flaws and controversies, such as resignations of moderators and resignation of senior community members over incidences of disrespect toward speakers of color around RustConf'23. Without minimizing the significance of these incidents, the author argues that Rust community dynamics actually reflect a higher degree of accountability than is typically observed in programming language communities; bad actors in the Rust community have often faced public repercussion for mistreatment of others, as opposed to mistreatment being swept under the rug. This is no small thing: it signals to marginalized members of the community that while absolute protection from social harm is not possible, the community response to any such harm would treat them as a person of value. Though much work remains to be done, Rust is chosen in part because the author deems the most welcoming branches of the Rust community capable of growing into the dominant force of the community and providing a welcoming environment for students over time.

Basic Rust Programs

We begin the Rust primer proper by exploring the basic building blocks of a program. The vocabulary for these building blocks is consistent across a wide range of programming languages: values are the results of computations, expressions are computations which potentially evaluate into results, definitions are parts of a program which introduce names as opposed to just performing a computation, and statements are programs which are executed for side effects such as changing program state instead of just evaluating to a value.

Values

A value is the most basic building block of a program. A value represents pure data and thus does not need to be executed any further. If we think of a program as instructions for answering some question, then a value is an answer. The notations for basic values tend to be similar across most programming languages, but it is important to discuss values explicitly because the word “value” is essential technical vocabulary which occurs frequently. In Rust, basic values include Boolean literals true and false, integer literals such as 0, -1, and 3000, string literals like “my string”, and floating point literals like 0., 303.12, or -101.0. Note that Rust allows floating-point literals which end in a period, with

no fractional part following them. As we introduce new types such as structures and enums, we will introduce new values for those types. Variables such as x are not values, but they do stand in for their values, making them closely related. When we wish to discuss an arbitrary value in the text, we will use the letter v . This letter v should be understood as a mathematical variable that ranges over different programs, not the specific Rust program v , which is a Rust variable.

References

References are how Rust implements the core idea of indirection, which appears in most programming languages in several different forms: pointers in C and C++, objects in Java and C#, or references in Rust, the ML family of languages, and elsewhere. To explain references, we explain their core operations: reference creation and dereferencing. If \boxed{x} is (for example) a variable, then $\boxed{\&v}$ is a reference to that variable; this is reference creation. If \boxed{r} is a reference to some value \boxed{v} , then $\boxed{*r}$ computes the value \boxed{v} ; this is dereferencing. To understand references, we also wish to understand whether $\boxed{\&x}$ is a value and, if not, what value it computes. It is not correct to call $\boxed{\&x}$ a value, because a value should be the same in all conditions: the number $\boxed{5}$ is always the number $\boxed{5}$, unchanging. A reference, however, indicates the

location at which a value is stored in a computer's memory, so it typically changes every time a program is run. In essence, the value of `&x` is an integer representing the memory location where `x` is stored. Yet, if `x` is located at address 123,456,789, then the value of `&x` (let's call that value `r`) is not the same as the integer literal `123,456,789`. References in Rust are safe, and it is essential to safety that references can not be interchanged with numbers and do not allow the same operations as numbers do, such as addition or multiplication. This is the fundamental difference between safe references in languages like Rust and pointers in C and C++: a pointer supports pointer arithmetic, which enables unsafe attempts to access memory locations that do not contain the intended data or do not even exist. In support of safety, there is also no way to write down a reference value directly in Rust source code.

Across programming languages, references serve several key functions such as being a key building block of data structures and allowing structures to be passed efficiently without copying. Though Rust does use references and similar constructs for these purposes, there is an additional key purpose in Rust: references are a major tool for making it simpler to write code that obeys Rust's requirement that each variable is used at most once. A proper discussion of that usage can only be undertaken after

introducing more features of Rust, and is thus deferred to the section on Ownership, Borrowing, and Scope.

Expressions

An expression is any program that can be evaluated. To evaluate a program is to execute it and, if it terminates, get back a value as its result. Expressions and evaluations are core concepts across most programming languages, and they are distinguished from other program building blocks such as statements and definitions by the fact that their results are values, specifically. Expressions are related to values in the following way: every value is an expression because it evaluates to itself immediately, but most interesting expressions are not values, most programs need to execute for some number of steps before returning their value. We will use letter e to mean “some expression e , it could be any expression,” i.e., it is a mathematical variable that ranges over expressions, as opposed to a Rust variable.

Many of the basic expressions are common among most mainstream programming languages, but it is important to mention them for completeness. The built-in types have operators that can be applied to them to form new expressions, such as:

- * the numeric operators $+$, $-$, $*$, and $/$,

- * the comparison operators `<=`, `<`, `==`, `!=`, `>`, and `>=` (where equality tests can be applied for a wide range of types, not just numbers), and
- * the Boolean operators `&&`, `||`, and `!`

Another indispensable expression is the function call. Let `f` be the name of a function and `args` be a comma-separated list zero or more expressions, then `f(args)` evaluates all the arguments to values, plugs in the values for the parameter variables of `f`, and executes the body of `f`, whose value is returned. The following example introduces a squaring function and uses it to compute a squared distance:

```
fn square(n: i32) -> i32 {
```

```
n*n
```

```
}
```

```
fn squared_distance(x1: i32, x2: i32, y1: i32, y2: i32) ->
i32 {
```

```
square(x1-x2) + square(y1-y2)
```

```
}
```

Rust contains two other syntaxes that are closely related to function call syntax: macro call syntax and method call syntax.

Macros are like functions that get executed at compile-time to generate custom code which will be executed at runtime. Macros are powerful enough to implement entire new programming languages inside of Rust. This chapter does not teach you how to implement new Rust macros, but it is important to be aware than macro calls use distinct syntax: every macro name ends with an exclamation mark (called like `my_macro!(x, y)`), so that they are easily detected by the parser, and some macros delimit their by square brackets (called like `my_macro![x, y]`), for flexibility of syntax. One example of a commonly used macro is the `format!` macro, which generates a formatted string using the format and parameters of the programmer's choice. It serves a similar role to `printf` in C, but uses a different notation for the first argument (called a `format string`), where braces `{}` stand in for each successive parameter. For example, the expression `format!("The {} I'm learning is {}", "language", "Rust")` returns the string "The language I'm learning is Rust". The `format!` macro is a perfect example of where macros are useful: the number of arguments and types of the arguments cannot be known without looking at the format string, and thus if `format!` were treated as a function, it could not possibly be given one correct type that applies for all format strings. In fact, languages like C typically must make their

formatting functions *unsafe* because their type systems cannot check a format string against the arguments, but a Rust macro can, by inspecting each macro call one-at-a-time. Conversely, macros cost complexity, and should not be used when functions suffice.

The *method call* notation does not represent a fundamentally different feature from function calls, but allows writing them in a style familiar to object-oriented programmers. The method call syntax `e.f(e1, ..., eN)` is equivalent to the call `f(e, e1, ..., eN)`. It is important to be familiar with both notations because documentation and sample code will often use method call notation. For example, the absolute value of x will typically be written as `x.abs()`.

Tuple expressions `(e1, ..., eN)` are used to package the results of several expressions together, for example the expression `(f(x), g(y))` would evaluate to a pair of values `(v1, v2)`, itself a value, where `v1` is the value of `f(x)` and `v2` is the value of `g(y)`. Tuples are used when the number of elements is fixed (e.g., 2 or 3) but the types of each element may differ from one another. The elements of a tuple can be extracted using numbers as method names: `e.0` extracts the first element of `e`, `e.1` extracts the second element, and so on. See the Patterns section for a cleaner way to extract elements, however. Tuples are

immutable, i.e., once a tuple has been created, its elements cannot be modified.

Arrays, like tuples, package several expressions together, but they are used when all elements have the same type and are mutable.

Arrays are created using square brackets instead of parentheses: `[1, 2, 3, 4]` is a four-element array of i32's. Elements of arrays are accessed using square brackets: `e1[e2]` computes the value v of e2, then computes the v'th element of array e1, starting from index 0. For the sake of space efficiency, the array type imposes a harsh requirement: the length of the array must be known at compile-time. This is rarely the case in our usage, so we will almost exclusively use the vector type, which is like an array except that its length can be determined at runtime, i.e., the vector stores its own length. Elements of vectors are looked up identically to arrays, but new vectors are created with a macro `vec!`, for example `vec![1, 2, 3, 4]` creates a vector of four i32's. The `vec!` macro is traditionally called using square brackets, to mimic array notation.

Conditionals (if-then-else) are a ubiquitous feature across programming languages, but actually come in two different versions depending on the language: if *statements*, which return no value, or if *expressions*, which return a value. In Rust, `if` is an expression, which means that its result can be built up into larger

expressions. Consider the following function `l1_norm`, which implements a common measure of how “big” a point is:

```
fn l1_norm(x: i32, y: i32) -> i32 {  
    (if x > 0 {x} else {-x}) + (if y > 0 {y} else {-y})  
}
```

The key point of this example is that both arguments to `+` are themselves if-expressions. This can only be done when `if` returns a meaningful value, i.e., when it is understood as an expression rather than a statement. This design decision, though powerful, does not come for free. It means that the `else` clause is mandatory, which is the only way to ensure that the expression has a meaningful value in every case. Without the `else` clause, a compiler error is produced. Some languages provide both conditional expressions and conditional statements. In C and its relatives in particular, their `if` keyword implements a statement, but the ternary expression operator `e1 ? e2 : e3` provides a direct equivalent of the Rust expression `if e1 { e2 } else { e3 }`.

What if we could take the idea of conditionals and make it far more flexible than a simple if-then-else? This would be profoundly

useful because conditionals occur constantly in programming.

The `match` keyword, which implements pattern-matching, does exactly that. The general syntax is `match e { p1 => e1, ..., pn => en }`, which is understood as “first evaluate `e` to a value `v`, then check whether `v` fits each pattern `pi`. As soon as you find a pattern `pi` that matches `v`, evaluate the corresponding `ei` and return its value. The expression `e` is called the *scrutinee* because the pattern match inspects it, and each `pi => ei` is called a branch. Patterns are a core category of syntax, on par with values, expressions, definitions, or statements. To learn more about `match`, continue onward to the Patterns section. Before you move on, however, see if you can guess what the following expressions do:

```
match x > y {
```

```
    true => x,
```

```
    false => y,
```

```
}
```

```
match x {
```

```
    (x, y) => x + y,
```

{}

```
match x {
```

```
(x, _) => x
```

{}

Now continue on to Patterns and see if your guesses were correct.

Patterns

The syntax of patterns looks similar to the syntax of expressions, but the *meaning* of a pattern is deeply different from that of an expression. The meaning of an expression comes from its value. The meaning of a pattern (variable name: p) is defined in two parts:

- * Which values v “match” the pattern p?
- * For values v which do match pattern p, what variables does the pattern introduce and what values do those variables have?

Pattern-matching concisely combines the work of conditional branching and variable definition, making it far more general

than other generalized conditional constructs such as the `switch` keyword in C.

One of the simplest patterns is the variable pattern. If you write a variable name as a pattern, it will match every possible value and will define that variable name to be equal to the value. For example, the example expression

```
match 3 {
```

```
    x => x*x,
```

```
}
```

computes the number 9 by defining `x` equal to 3. This example expression is not a recommended use of pattern-matching. If you wish only to define a variable equal to a specific expression, use the `let` definition described in the Definitions section. Variables become powerful (only) when combined with the other patterns; we introduce variables first to highlight the concept that a pattern can define new variables. In the author's experience, readers should be cautious of a particularly common mistake in the use of variable patterns: a variable pattern *always* defines a variable, even if a variable with the given name already exists. In the code `match e1 { x => match (e2) { x => e3 } }`, the inner

match *does not* check whether e1 and e2 have the same value.
 Rather, it entirely ignores the value of e1 and makes x equal to e2.

The other simplest patterns are the literal (value) patterns. Many values, such as Boolean, integer, string, tuple, and array values can appear directly as patterns. A few literals, such as floating point literals, are not allowed in patterns because floating-point number computations involve rounding, which makes exact comparisons challenging.

The first example program

```
match x > y {
```

```
true => x,
```

```
false => y, }
```

uses Boolean patterns. It computes the value of expression `x > y`, then evaluates to x if the value was true, else y. In short, it computes the maximum of x and y. It is equivalent to the expression `if (x > y) { x } else { y }`. In general, every if-then-else expression could be written as a match expression:

`(e1) { e2 } else { e3 }` is always equivalent to `match e1 { true => e2, false => e3, }`. However, Rust maintains a separate syntax for `if` because it is so common.

The compiler performs important correctness checks on your pattern-matches, called *exhaustiveness* and *redundancy* checking. Exhaustiveness means every possible (well-typed) value of the scrutinee must match at least one pattern, and redundancy checking means that each pattern must match at least one value that was unmatched by all the previous patterns. For example, exhaustiveness checking prevents us from forgetting the `else` on an if-then-else) and redundancy prevents us from accidentally writing two false cases (equivalent to a nonsensical if-then-else with two else clauses).

As soon as we move to types with large numbers of values, such as integers, we need patterns that mean “everything else,” because we would not dare to write a separate branch of code for every integer.

The *wildcard* pattern, written as an underscore `_`, stands for “everything else.” It matches every value and introduces no variables. This makes it possible to make good use of integer patterns, such as the following function `flip_digit` which, if the argument is a single digit `d`, returns `9 - d`, else returns its argument:

```
fn flip_digit(n: i32) -> i32 {
```

```
match n {
```

```
 0 => 9,
```

```
 1 => 8,
```

```
 2 => , 7
```

```
 3 => 6,
```

```
 4 => 5,
```

```
 5 => 4,
```

```
 6 => 3,
```

```
 7 => 2,
```

```
 8 => 1,
```

```
 9 => 0,
```

```
 _ => n,
```

```
} }
```

The function `flip_digit` is long, so long that it causes concern, because the written description of its goal was much shorter. There are two kinds of patterns that alleviate this problem. *Range* patterns `min..=max` are specific to numeric patterns, and match any value `v` in the given range, i.e., where `min` `<= v` and `v <= max`. Thus, `flip_digit2` provides a more concise implementation:

```
fn flip_digit2(n: i32) -> i32 {
    match n {
        1..=9 => 9 - n,
        _ => n,
    }
}
```

Though ranges are elegant for integer programs, they do not generalize well to other types such as strings, or to non-adjacent numbers. *Or* patterns are less concise than ranges, but generalize better. The pattern `p1 | ... | pn` matches a value `v` if it matches any of the patterns `pi`. Or patterns are often introduced no

variables, but are allowed to introduce variables so long as every sub-pattern is consistent. We reimplement `flip_digit` with an or pattern in `flip_digit3`:

```
fn flip_digit3(n: i32) -> i32 {
    match n {
        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 => 9 - n,
        _ => n,
    }
}
```

Though `flip_digit3` is less concise than `flip_digit2`, or patterns are useful if you want special handling for different strings. As a toy example, the following expression computes location-specific slang for subs, a kind of sandwich:

```
match location_string {
    "Philadelphia" => "hoagie",
    "New York" => "hero",
```

```
_ => "sub",
```

```
}
```

This example, though toy, illustrates why one should be careful using the wildcard pattern: it is easy to miss some cases that need special handling. Many places other than Philadelphia and New York have their own terminology for these kinds of sandwiches!

Tuple patterns `(pat1, ..., patN)` only match tuples of form `(v1, ..., vN)` (i.e., those with N elements), and match them only if *every* value *v_i* matches the corresponding pattern *p_i*. If they do match, then `(pat1, ..., patN)` introduces all the variables introduced across every *pat_i* (where no two patterns should use the same variable names as one another). Tuple patterns are highly flexible when combined with other features such as variable, wildcard, and literal patterns. We return now to the second and third example pattern-matching programs from the Expressions section:

```
match x {
```

```
(x, y) => x + y,
```

```
}
```

The above example assumes x is a pair, defines x and y to be the respective elements of that pair and evaluates to the sum of the elements

```
match x {
```

```
    (x, _) => x
```

```
}
```

The above example assumes x is a pair, defines x to be the first element while ignoring the second, and evaluates to the first element.

Definitions

As we investigate the distinctions between statements and definitions, please be aware that other sources such as “The Rust Programming Language” use terminology differently: they do not distinguish definitions as a class of their own but instead consider them to be a strict subset of statements. This is not wrong in the case of Rust, but we introduce both terms as they are both important in the broader study of programming languages.

Definitions are one of the key building blocks of programs, separate from values and expressions. Definitions are

distinguished from other program building blocks by their ability to directly introduce names; this notion of definition is key technical vocabulary which will occur across most programming languages. In casual usage, definitions might be discussed interchangeable with their close relative, *declarations*. The difference between the two is that a definition introduces a name and determines its meaning at the same time, while a declaration introduces a name but does not immediately determine its meaning.

Names are a central aspect of programming languages, to the point that many mathematical models of programs such as lambda calculi and process calculi place names as one of their most central features. This chapter does not concern itself with technical implementation details of naming, but rather programmer-facing language features.

The fundamental definition required for every Rust program is the function definition, which follows the syntax `fn` `function_name` `(args)` `->` `return_type` `{ body }`. `fn` is a keyword, `function_name` is an identifier, `args` is a comma-separated list of identifiers with their types, `return_type` is the return type, and `body` is the function body. For example, the definition `fn double(n: i32) -> i32 { 2*n }` introduces the name `double` the function which accepts a 32-bit integer `n` as its

parameter and returns twice `n` (see the Types section for discussion of types and type definitions, another kind of definition). A function definition can optionally have the keyword `pub` before the keyword `fn`. This `pub` keyword makes the function definition callable from other files (modules); otherwise it is callable only within the file (module) in which it is defined. It is standard practice to leave out the `pub` keyword when writing a helper function, using it only when you know a function will be called from another file.

The second essential kind of definition is the `let` definition, which evaluates an expression to a value and assigns it a name given by the programmer. The let definition demonstrates that while expressions and definitions are distinct, they are closely related in the sense that definitions typically contain expressions within them. As an example, `let x = 5;` is a definition which defines the variable name `x` to stand for the value `5`. Though the example `let x = 5;` is quite modest, the let definition becomes the backbone of writing any complex function, especially when used repeatedly. Though truly complex code is challenging to fit in a textbook example, consider the function `dot__product`, which computes the dot product of two vectors of two elements each, as a simple example of how nested `let` definitions contribute to structuring function implementations:

```
fn dot_product (x1: i32, x2: i32, y1: 32, y2: i32) -> i32 {
```

```
    let prod1 = x1 * y1;
```

```
    let prod2 = x2 * y2;
```

```
    prod1 + prod2
```

```
}
```

A `let` definition has two optional pieces of syntax: a type annotation and the `mut` keyword. In a `let` definition of the form `let x : t = e;`, the type annotation `: t` indicates that the expression `e` is required to have type `t`, else a type error is produced at runtime. If the annotation is left off, the compiler still computes the type of `e` at compile-time and determines that it is used correctly for that type. For this reason, it is possible and even common to write large, complex programs without type annotations on `let`. As a rule of thumb, you should include the type annotation whenever it would aid in your understanding of the program, either by making it easier to read or by helping you debug a compiler error. Compared to type annotations, the `mut` keyword serves an entirely separate purpose: if the `mut` keyword is included immediately after the `let` keyword,

then an assignment statement (see Statements section) can be used to update the value of the variable later, otherwise the value cannot be changed. This language design decision, which could be called immutability-by-default, is intended to reduce the risk of bugs involving mutation, without prohibiting mutation entirely.

Statements

Statements (variable name: s) are one of the key building blocks of programs, on par with values, expressions, patterns, and definitions. Statements typically contain expressions or definitions within them, but are distinct from expressions. Like an expression, an expression can be executed, but unlike an expression, it is not associated to a value, thus we use the word execution rather than evaluation. Because a statement produces no value, its meaning is defined by

- * Any variables defined by definitions contained within the statement and
- * Any *side-effects* of execution, defined as observable behaviors other than evaluation. Typical examples of side effects include mutating the value of a mutable variable and reading input from a file or user, writing output to a file or user. Looping forever is also considered a side effect, albeit one which can be achieved even with expressions alone.

Every expression `e` can be made into a statement `s` in two distinct ways. Adding a a semicolon produces the statement `e; e;` evaluates `e` (presumably for its side effects) and ignores the value. In contrast, `return e` evaluates `e` and immediately returns its value as the return value of the current function. The `return` keyword is only required for an early return; if the body of a function ends in expression, Rust understands it as the return value of the function.

Assignments, written `x = e;`, are one of the most common statements. If `x` is a mutable variable (defined using the `mut` keyword), then `x = e;` evaluates `e` to a value `v` and updates the value of `x` to be `v`. If `x` is not defined as mutable, then assignments to `x` do not compile. The left-hand side of an assignment can be more than a variable name; the expressions which are allowed to appear on the left-hand side of an assignment are called *lvalues* for short. When `e1` is a reference, then `*e1 = e2` updates the reference to refer to the value of `e2`. When `e1` is an array and `e2` is an integer, then `e1[e2] = e3;` updates the `e2`'th element of `e1` to be the value of `e3`.

Input and output, which are fundamental to interacting with humans, are performed using functions and macros. The macro `println!` is used to print text to a command line, using the same syntax as the `format!` macro. Conversely, consult the

documentation for `std::io::stdin` to learn about the `read_line` function for reading input from a user on the command line.

Loops are statements, though not a side effect in their own right. A `while` loop, written `while e { s }` repeats the statement `s` as long as the Boolean expression `e` evaluates to the Boolean value `true`.

The `for` loop is more complex because it can loop over any *iterator*, a concept that includes both numeric ranges and the elements of arrays or vectors. The general form is `for x in iterator { body }`. The following numeric example prints out all numbers from 1 to 10, inclusive:

```
for i in 1..=10 {  
    println!("Number: {}", i);  
}
```

To loop over an array or vector, we call the `.iter()` method to access its iterator. The following statement computes the sum of elements in an array `e`:

```
let mut sum = 0;  
  
for x in e.iter() {
```

```
sum = sum + x;
```

```
}
```

Note that the value `x` refers to the values of the array, not the indices. This complicates some use cases such as iterating over the values of two arrays in lockstep. For that usecase, use the `zip()` method of an iterator. For example, the following statement computes the dot product (the sum of pairwise products) of two arrays `e1` and `e2`:

```
let mut sum = 0;
```

```
for (x, y) in e1.iter().zip(e2.iter()) {
```

```
    sum = sum + x * y;
```

```
}
```

Types

Though we have carefully minimized discussion of types (variable name: `t`) in the chapter thus far, types are an essential aspect of Rust. Rust is a statically-typed language, meaning that the type of an expression `e`, determined at compile-time, is a *guarantee* about

e: if e evaluates to some value v, then it will be a value of type t.

All of the rules about types for a given language, taken together, are known as a *type system*. Static type systems like Rust's increase the frequency of compiler errors by checking types at compile-time but, in so doing, prevent those errors from hiding from the programmer's view until runtime.

Basic Types

The Boolean type `bool` is the type assigned to the values `true` and `false`.

Floating-point numbers come in two types: `f32` and `f64` with 32 and 64 bits respectively.

Rust has multiple integer data types, which vary both in the size of the integer in memory and whether negative integers are included (signed integers that can be negative, or unsigned integers which cannot be negative but can represent higher positive values). Sizes range from 8 bits to 128 bits, with the signed integer types of each respective size named `i8,i16,i32,i64`, and `i128`, and the unsigned types of each respective size named `u8,u16,u32,u64`, and `u128`.

Of Rust's many types for strings, the most relevant are the string slice type `&str` and the growable string type `String`. Both types

track the length of the string, but only `String` is suitable for mutable code. Because string literals are not to be mutated, they have type `&str`. In the case that you wish to build a larger string from a string literal, you can use the method `.to_string()` to convert it to the `String` type for additional processing. Leaving off this method call produces a compiler error.

All of the basic types are governed by typing rules, rules that say how each expression can be used. A program which follows all the rules is well-typed, all others are ill-typed.

Compound Types

A compound type is any type built up from smaller ones. These include both built-in features like tuples, arrays and vectors, and references, as well as well as user-defined types such as structs and enums.

Tuple type syntax mirrors tuple expression syntax: a tuple `(e1, ..., eN)` has type `(t1, ..., tN)` when each e_i has type t_i , this is the typing rule for tuples. Tuple projection $e.i$ has type t_i when e has type `(t1, ..., ti, ..., tN)` for some $N \geq i$. Projection is ill-typed when $N < i$.

There are several distinct types relating to arrays. The array type proper is written `[t;n]` meaning “an array of n elements, each of type t.” The array type requires the length of the array to be known at compile-time, which greatly restricts usage, yet enables a more space-efficient representation at runtime. Its close relative, the slice type `[t]`, refers to an array of elements whose length is not known at compile-time. Instead, the slice tracks the length at runtime, sacrificing space for flexibility. One example of slices’ flexibility is that they provide an elegant way to manipulate arrays’ ownership, e.g., an array could be partitioned into two or more slices, which own distinct elements of the array that can be manipulated independently. Neither arrays nor slices allow the size of an array to grow dynamically. For that, we use the type `Vec<t>`, meaning vector of any length, with elements of type t. The angle brackets indicate a *type argument*; the `Vec` type can be used for any type of elements. The `Vec<t>` type is given to the vectors created by the `vec!` macro.

References and Borrowing

The types for references closely follow the syntax: immutable reference type `&t` indicates a reference that can be used to read a value of type t, but not modify it, while the mutable reference type `&mut t` carries the ability to both read and write. The typing rules are simple enough: if e has type t then `&e` has

type `&t` and `&mut e` has type `&mut t`, where `&mut e` is only allowed if `e` is defined as mutable. For references `r` of either type `&t` or `&mut t`, `*r` has type `t`. Despite these simple-enough typing rules, references introduce what has been considered the most complex aspect of Rust's type system for new learners: the borrowing system, implemented by the *borrow checker*.

Recall from the References subsection that most values in Rust can only be used once, e.g., passing an argument directly to a function destroys the ability to use the argument again in the future. Though essential for safety and speed, this limitation is too strong, and references are used to circumvent the limitation by borrowing ownership of a value temporarily and giving it back at the end, yet borrowing must not be made so strong as to violate safety of Rust's affine type system. Let's consider different ways that references could be used, and how they might conflict with affinity, the property of being affine. The example below demonstrates an unacceptable use of *mutable* references:

```
let mut x = 5;
```

```
let r1 = &mut x;
```

```
let r2 = &mut x;
```

```
*r1 = *r2 + 1;
```

A reference represents temporary (borrowed) ownership of the value it refers to. In this case, r1 and r2 purport to own x at the same time. This certainly violates the requirement that each value has a single owner and thus violates affinity. A modest stretch of the imagination shows how similar errors would cause crashes at runtime: if memory is freed when a reference that owns x goes out of scope, then a double-free memory bug would occur.

Immutable references carry their own assumptions which could conflict with mutable references: if I own an immutable reference, I expect its value to never change. If mutable and immutable references were mixed, the following unacceptable code might occur:

```
let mut x = 5;
```

```
let imm = &x;
```

```
let r = &mut x;
```

```
let x1 = *imm;
```

```
*r = x + 1;
```

```
let x2 = *imm;
```

Here, we would expect x1 and x2 to have the same value as one another, yet `*r` and `*imm` are the same, so that x1 is 5 and x2 is 6.

In contrast to these first two examples, however, multiple immutable references can coexist peacefully:

```
let x = 5;
```

```
let ir1 = &x;
```

```
let ir2 = &x;
```

```
f(ir1,ir2);
```

When we pass ir1 and ir2 to the same function f, we should hesitate. Surely, this breaks affinity because we duplicated x. However, in a sense that one could rigorously prove, it does not break affinity in any way that truly matters. Immutable references are referentially transparent, meaning that the expressions `*ir1`, `*ir2`, x, and 5 are completely indistinguishable and have the same behavior in all contexts. Likewise, a program which produces multiple immutable references is indistinguishable from one which produces a single immutable reference and reuses it.

Though one should expect to struggle with borrow-checker errors when writing their first Rust programs, we summarize the discussion with a useful mantra: “*When trapped by the borrow checker, WOoRM your way out!*” The acronym WOoRM stands for “Write One or Read Many,” and means that at any moment in time it is acceptable to have either a single mutable reference to given value *or* an arbitrary number of immutability references, but never both. Most borrowing errors can be boiled down to the WOoRM mantra. If the WOoRM mantra cannot save you, you may have found a case where you truly need to perform a deep copy of your data using its `.clone()` method, which is significantly less efficient but creates two truly independent values with separate ownership.

The smart pointer type `Box<t>` is thematically related to references, but does not factor into borrow-checking. A `Box<t>` is created by calling `Box::new(e)` where `e` has type `t`. An expression `e` of type `Box<t>` is dereferenced by writing `*e`. For more details on the syntax `::` that appears in `Box::new(e)`, see Complete Rust Programs.

The `struct` keyword is the first major way for programmers to introduce new types. Structs, like tuples, store several values together, but they differ in that they give a name to every element and a name to the struct type as a whole. In practice, tuples are

used when the elements are well-understood from context, e.g., when there are only a few elements or the elements have an intuitive ordering. Structs are used when there are more elements, when the order is not clear, or when there is a desire to reuse the same type many times.

To introduce a new struct type, use a struct definition, which is a type definition. The syntax `struct Name {f1:t1, ..., fN:tN}` introduces a new type named Name with N fields, each field respectively named fi of type ti. Like function definitions, all type definitions can optionally begin with the keyword pub to make them usable in other files. For a canonical example from the Rust reference, 2-dimensional i32 points are defined `Point {x: i32, y: i32}`. Values of struct types such as Point are created using the type name, e.g., `Point{x:5, y:10}` constructs a point from coordinates 5 and 10. Fields are extracted from structs using the field names, e.g., if p is a Point then p.x is the x coordinate.

The other major way for programmers to define new types is with the enum keyword. The enum keyword is significantly more general than the corresponding features in languages such as C. It is a powerful tool for building any type definition which must be broken into different cases, including many recursive data structures. It is the type system feature that implements the idea of “or”.

The simplest enum definitions only enumerate a list of constant values, for example:

```
enum InkColor {
```

```
    Cyan,
```

```
    Magenta,
```

```
    Yellow,
```

```
    Black,
```

```
}
```

The InkColor type has exactly four values,

written `InkColor::Cyan`, `InkColor::Magenta`, `InkColor::Yellow`,

and `InkColor::Black`. This type is no more technically sophisticated than a two-bit integer. Rather, the power of enums lies in the capacity to put arguments on each variant (i.e., branch) of the definition. The following enum InkInventory represents the inventory at a print shop:

```
enum InkInventory {
```

```
SingleColorCartridge(InkColor, i32, String), // color,  
quantity, model number
```

```
TriColorCartridge(i32, String) // quantity, model number
```

```
RefurbishedCartridge(InkColor, i32) // color and quantity, no  
model number
```

```
}
```

an item in the InkInventory can be one of three different things: a single-color cartridge with its color, quantity, and model number, a tri-color cartridge with its quantity and model number, or a refurbished cartridge whose color and quantity but not model number are tracked. Though this definition allows us to model highly heterogenous data, it is concerning that the names of each field were put only in comments and not given field names as in the struct. In actuality, both enum and struct keywords give the programmer the option to define a type with positional elements (using parentheses as in our InkInventory example) or named fields (using braces and colons as in our Point example).

The full power of enum comes in defining inductive (colloquially called recursive) types. Linked lists are a common example, so let's attempt to define them ourselves. For simplicity, we restrict

the elements to integers. The follow attempt seems promising, but is rejected by the compiler:

```
enum IntLinkedList {
```

```
    Empty,
```

```
    Cons(i32, IntLinkedList),
```

```
}
```

In this definition, `Empty` is the empty list and `Cons(x, L)` is the list that starts with `x` and is followed by `L`.

The error arises from the inductive reference to `IntLinkedList`. Rust takes a simplistic approach to laying out data in memory: if we write `Cons(i32, IntLinkList)`, Rust interprets this as a request to store the next linked list value (`Cons` or `Empty`) directly inside the first `Cons` value. In short, Rust understands this code as declaring the size in memory of a single `Cons` value to be variable, which is rejected. The standard solution is that every inductive appearance of the enum you are defining must appear under a `Box<>`. The `Box` type represents a smart pointer, which always has a fixed size, no matter how complex the underlying data. In simpler words: implementing inductive types requires references

for indirection, just as a pointer would be used to implement them in C. The addition of a Box makes Rust accept the definition:

```
enum IntLinkedList {    Empty,    Cons(i32,  
Box<IntLinkedList>), }
```

Pattern matching now allows us to write concise recursive functions over inductive types, such as this function for the length of an IntLinkedList (note that the name of the enum type is required in the pattern match):

```
fn ill_length(L: Box<IntLinkedList>) -> i32 {
```

```
    match L {
```

```
        IllLinkedList::Empty => 0,
```

```
        IllLinkedList::Cons(_, xs) => 1 + ill_length(xs),
```

```
    } }
```

The builtin types of Rust can be used in a variety of ways such as comparing for equality, copying them with the `.clone()` method, and displaying them as strings. How do you get these features for your own type definitions? The derive keyword allows these

features (traits) to be implemented in many cases. To automatically derive implementations of traits, write `[#` `[derive(TraitList)]` on the line before your type definition. For example, the trait list `#[derive(Hash, Eq, PartialEq, Debug, Clone)]` derives the ability to store your type in a hash table, compare it for equality, produce a string representation suitable for debugging, and clone (deep copy) it. The Debug trait allows printing your own data with the `format!` and `println!` macro, though it requires using `{:?}` in the format string rather than `{}`.

Complete Rust Programs

We have introduced all the core features needed to write a file full of Rust code, and all that remains is to explain the relationship between files and full programs. In complex Rust programs, the fundamental organizational unit for definitions is a `module`: code can view all definitions in its own module but only public definitions from other modules, and the same definition name may be used more than once as long as it is used in different modules. Every file of Rust code gets its own module, and this is sufficient for most readers' uses. If you wish to introduce additional modules within a single file, you can do so by wrapping the module's code in the syntax `mod modulename { ... }`. Much more frequently, you will write a module name when writing

down which module a given function belongs to, such

as `Boxnew(e)` when creating a new boxed value.

The `use` keyword is the other core component of the module system. The top of every Rust file typically contains `use` definitions which determine which modules from external libraries can be used in the file, such as

```
use stdhash::Hash;
```

```
use rpds::HashTrieMap;
```

to use the standard Hash module and the custom HashTrieMap data structure from rpds.

To complete our Rust programs, all we need to do is define a `main()` function, the function where program execution starts. The main function is defined with no argument and no return type annotation:

```
fn main () { body }
```

Note that if you wish to access command-line arguments to a Rust program, you do so through the standard library (`std::env`) instead of as arguments to the main function.

Tooling

Rust has been praised for the quality of its tooling: the software you interact with in the process of developing Rust programs, such as code editors, build management, and package management.

Like most programming languages, Rust does not force the programmer to use a specific code editor. However, VSCode is commonly recommended because it is popular in its own right and has a robust extension for Rust programming.

See <https://code.visualstudio.com/docs/languages/rust> for installation instructions both for VSCode proper and for the Rust extension, which is named `rust-visualizer`.

The build management system and package manager are implemented through a common program named `cargo`, accessible by command-line. When developing a new program in Rust, it is common to create the project by running the command `$cargo new projectname` on the command line, which creates all files required by cargo. In cargo terminology, a Rust project is called a `crate`, e.g., when you compile your project, you are building a crate, and when cargo downloads a package you depend on, it is downloading a crate. If you wish to compile and run your crate from the command line rather than in VSCode, you can do so with the command `$cargo run`.

The configuration for your crate is stored in a file called `Cargo.toml`. This file is often concise. Consider a `Cargo.toml` file created for a companion assignment to this book:

```
[package]
```

```
name = "asgn3"
```

```
version = "0.1.0"
```

```
edition = "2023"
```

```
[dependencies]
```

```
rpds = "0.13.0"
```

In this `Cargo.toml` file, the first four lines are auto-generated and simply give a name and version number to the project. The only manually-added line was the last one, which specifies which version of the `rpds` library to use. Cargo automates behind-the-scenes steps required to use that library, such identifying dependencies of `rpds` and downloading and compiling all dependencies. The Cargo system also comes with substantial capacity to generate documentation. As such, reference documentation for all published crates is available

through <https://crates.io/>. Readers are recommended to consult crates.io to learn more about any crates used in their work.

Related Work

The book “The Rust Programming Language” was a central source for the development of this chapter, but the style and scope differ significantly. The style of this chapter is meant to introduce Rust in a way that emphasizes key terminology about programming languages, i.e., it aims to prepare you for the discussions you will undertake throughout a book on programming languages.

Because “The Rust Programming Language” is a book-length treatment of a single language, it can address that language in far greater depth, and is thus recommended as an additional as-needed reference, in particular because it is available online free-of-charge. In contrast, we limit this Rust primer to a single chapter because it is meant to provide more specific and instrumental knowledge: what does the reader need to know in order to complete programming language exercises in Rust? In addition to “The Rust Programming Language,” readers are invited to use other official Rust resources as supplemental material: Rust By Example provides example-driven practical knowledge that is suitable for the role of Rust in this book, the Rust Playground provides an easy-to-access programming environment. Readers should consult the Rust standard library

documentation to understand which operations and functions are available for specific types that arise in exercises.

A secondary reason that Rust is an enriching choice of language for an academic context is the active research community surrounding it. On one side of the research community, there are numerous research projects which investigate what academic programming languages can be implemented on top of the Rust type system, how Rust language tools should be developed, or how its compiler can be improved. For students with future academic wishes, it is worth learning a language used in that research community. On the second side of the research community, Rust is emerging as a language of interest within the computer science education community, including but not limited to the Rust-Edu workshop, starting in 2022, whose proceedings are publicly available. In that workshop, the present author explored the tradeoffs of using Rust in a first programming course. Other relevant works at the Rust-Edu workshop included a talk by Galileo Daras in support of an “intersectional” approach to using Rust in education, consistent with this book’s mission of teaching programming languages through the lenses of multiple disciplines, and the RustViz tool for developing diagrams about borrowing, which emerged as a consensus example of a difficult-to-learn Rust feature. Other talks presented the recent emergence of the first Rust-based courses at Northeastern University, Fudan

University, Rochester Institute of Technology, and an unnamed European University, highlighting a growth in Rust interest across at least three continents.

Understanding Compiler Errors

Every programmer will encounter errors in their program, and in a statically-typed language like Rust, many of those errors will come from the compiler. Though frustrating, compiler errors are there to help you by catching potential bugs early. Learning to read and respond to compiler errors is a key programming skill. To that end, we explore some common compiler error messages that new programmers may encounter. We start by exploring messages from incorrect programs mentioned in this chapter.

Try out the following program:

```
enum IntLinkedList {
```

```
    Empty,
```

```
    Cons(i32, IntLinkedList),
```

```
}
```

which will produce a message like “recursive type `IntListList` has infinite size” and recommend “insert some indirection (e.g., a `Box`, `Rc`, or `&`) to break the cycle: `Box<`, `>`”. In this example, the compiler provided perfect advice: Using `Box<IntLinkedList>` in `Cons` solves the issue.

Let's try out the following program:

```
let mut x = 5;
```

```
let r1 = &mut x;
```

```
let r2 = &mut x;
```

```
*r1 = *r2 + 1;
```

The error message is “cannot borrow `x` as mutable more than once at a time, second mutable borrow occurs here” followed by the locations of the first and second borrows. It accurately reports the problem but does not recommend a solution. Recommended solutions include double-checking that both references need to be mutable and, if so, performing a `clone()` before forming the references.

Let's now consider a different borrowing error:

```
let mut x = 5;
```

```
let ir = &x;
```

```
let r = &mut x;
```

```
let x1 = *imm;
```

```
*r = x + 1;
```

```
let x2 = *imm;
```

This time the error message starts with “cannot borrow `x` as mutable because it is also borrowed as immutable” and gives the locations. As before, the solution is to make all the mutable reference immutable if possible, limit the scopes so that they are not defined at the same time, or perform a clone.

Let’s now explore some errors that we could have made if we wrote the code in this chapter a bit differently. The following alteration of `ill_length` forgets the type name in the pattern match:

```
fn ill_length(ill: Box<IntListList>) -> i32 {
```

```
match *ill {
```

```
Empty => 0,
```

```
Cons(x, xs) => 1 + ill_length(xs),
```

```
}
```

Here the error message is “cannot find tuple struct or tuple variant `Cons` in this scope” which is less direct than before.

The compiler recommendation is consider importing this tuple variant: `use crate::IntListList::Cons;` which would be a solution if Empty is also imported, but is not preferred style in all cases; the fix is to include the names in the patterns.

If we have too few cases, such as a missing base case, we get a different error:

```
fn ill_length(ill: Box<IntListList>) -> i32 {
```

```
match *ill {
```

```
    IntListList::Cons(x, xs) => 1 + ill_length(xs)
```

```
}
```

The error message “non-exhaustive patterns:

` `IntListList::Empty` not covered, the matched value is of type ` `IntListList` " and recommendation "ensure that all possible cases are being handled by adding a match arm with a wildcard pattern or an explicit pattern as shown:
` , IntListList::Empty => todo! () " are accurate. In this case, the solution is to add the Empty case; in some circumstances, wildcards would be appropriate.

A duplicate branch leads to a different warning:

```
fn ill_length(ill: Box<IntListList>) -> i32 {
```

```
    match *ill {
```

```
        IntListList::Empty => 0,
```

```
        IntListList::Empty => 0,
```

```
        IntListList::Cons(x, xs) => 1 + ill_length(xs)
```

```
} }
```

The message “unreachable pattern” means one pattern is dead code because it is redundant. The solution is to delete it.

A wide array of compiler errors fall under the pattern that one type was found where another type was expected. To produce such an error, we can simply forget a dereference:

```
fn ill_length(ill: Box<IntListList>) -> i32 {
```

```
    match ill {
```

```
        IntListList::Empty => 0,
```

```
        IntListList::Cons(x, xs) => 1 + ill_length(xs)
```

```
}
```

Resulting in the message “mismatched types, expected struct `Box<IntListList>`, found enum `IntListList`” and recommendation “consider dereferencing the boxed value: `*ill`”. Because either the expected or found type can be at fault, this recommendation is not always reliable, though it was correct in this case.

If we make a syntax error, such as leaving off a comma on a match arm, the error will be marked as a syntax error:

```
fn ill_length(ill: Box<IntListList>) -> i32 {
```

```
match *ill {
```

```
    IntListList::Empty => 0
```

```
    IntListList::Cons(_, xs) => 1 + ill_length(xs),
```

```
}
```

The error messages “missing a comma here to end this `match` arm: `,`” expected `,` following `match` arm” and “missing a comma here to end this `match` arm: `,`” accurately assesses the missing syntax and its location.

You will surely encounter other compiler errors in your work, but practice makes reading error messages easier. The take-away message from these examples is that when the Rust compiler error includes a recommendation on how to fix the error, you should try the recommended action. It does not always work, but often does.

Classroom Activities:

1. Explore the official Rust documentation together with students

2. Help your students scaffold how they should organize information in their notes and mind: let them know that to learn a new type, the key aspects are learning the operations on that type, the values on that type, and the typing rules. Syntax, while also important, is also a means to the end of understanding how the type operates.
3. Draw a Venn diagram of the relationship between strings of text, syntactically valid programs, well-typed programs, well-typed programs that evaluate to values, and values.
4. Draw a diagram of the relationship between expressions, statements, definitions, patterns, and values.
5. Select several programming exercises which you do not intend to assign to students and live-code those exercises in class.
6. Present example code snippets that feature borrowing and use a RustViz-style borrowing diagram to explain borrowing.

Exercises:

1. This lecture introduces lots of new words. Make yourself a review sheet that defines core terms such as: expression, value, definition, statement, statically-typed, and another key terms that are important to you.
2. Discuss with a friend whether you and they find the same terms to be key

3. Make a list of all Rust features discussed in this chapter.

Which ones are must-haves for every programming language with large programs? Now consider the rest: these are likely where Rust's design novelty lies. Revisit this exercise when designing your own language: where are you implementing general-purpose features and when are you implementing *your* design?

4. Keep a diary of experiences programming in Rust. This is a form of *autoethnography*: recording your own lived experiences for research purposes. During every coding session, add an entry to your journal, which tracks the following: What operators or functions did you look up? What error messages confused you or didn't confuse you? How much of your time was spent thinking vs. coding vs. debugging? What bugs, if any, consumed the most of your time? What about Rust did you learn during this session? In that experience, what made you look forward to learning more or coding more? What drives you away, and what do you not yet understand?

5. Look up the documentation pages for the following: `Box::new`, HashTrieMap, and format!. Which of them are in the Rust reference documentation and which are on crates.io? For each of the documentation pages, write down one thing you learned from the documentation page that was not in this chapter.

6. Find the syntax error in each of the following strings:

- a. `fun main () { }`
- b. `for (x = 0; x < 10; x = x + 1) { println!("{}", x); }`
- c. `match (1,2) { (1, _) => 3 (_ ,2) => 3 }`
- d. `let mut x = false; if (x = true) {5}`
- e. `let mutable x = 10; x = 3;`

7. Find the type error in each of the following:

- a. `1 + 3.2`
- b. `true || 1`
- c. `if 1 then {2} else {3}`
- d. `let mut x: i32 = 1; let y:i64 = 2; x = y;`
- e. `let mut x: i32 = 1; let y:i32 = 2; y = x;`
- f. `let b = true; match &b { true => 3, false => 5 }`
- g. `let mut x = 1; let r1 = &mut x; let r2 = &x; *r1 + *r2`

8. Explore the website RosettaCode.org, which features the same programming problems solved in a wide array of languages. Pick three problems that seem interesting to you and which have been solved both in Rust and a common language. Look at the implementation in the common language for

inspiration, then write your own solution in Rust, then compare their solution to yours. What, if anything, can you learn from their solution?

9. Implement the following functions. For the following, assume the definitions

```
enum DayOfWeek  
{Sunday, Monday, Tuesday, Wednesday, Thursday, Friday,  
Saturday, Sunday}  
and enum Tree {Empty, Node(Box<Tree>,  
i32, Box<Tree>), } where the latter is used for binary search  
trees:
```

- a. fn nor(b1: bool, b2: bool) -> bool // true if neither arg is true
- b. fn Linf_norm(x:Vec<f64>) -> f64 // maximum element of vector
- c. fn reverse(x:Vec<bool>) -> Vec<bool> // reverse order of elements
- d. fn concat_all(x:Vec<String>) -> String // concatenate all strings
- e. fn day_name_string(d:DayOfWeek) -> String // string for day
- f. fn contains_sunday(x:Vec<DayOfWeek>) -> bool // whether has Sunday

- g. fn tree_size(t: Tree) -> i32 // number of Node values in tree
- h. fn tree_depth(t: Tree) -> i32 // max number of Nodes from root to leaf
- i. fn insert(t: Tree, k: i32) -> Tree // insert t in sorted order
- j. fn remove(t: Tree, k: i32) -> Tree // delete k, no change if not present
- k. fn contains(t: Tree, k: i32) -> bool // whether k appears in any node

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

*bookish.press/book/chapter4
Chapter 4*

Regular Expressions

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * The Implementer cares about regular expressions because they are a core tool for implementing a programming language, specifically the part that parses basic building blocks of syntax
- * The Practitioner cares about regular expressions because they are also used in a wide variety of small text-processing tasks in programming, such as parsing email addresses, IP addresses, dates and times, passwords, plaintext data files, and more. They can also be used to efficiently search through large bodies of text, such as the code you write.

- * The Theorist cares about regular expressions because their simplicity is their strength. Regular expressions are simple enough that many properties about them can be computed by programs, making it possible to design algorithms and tools that make the Implementer's and Practitioner's jobs easier.
- * The Social Scientist and Humanist may be curious about the role of regular expressions in programmer culture, where they have become the basis of programming "games" such as "regex golf".

What is a regular expression (often abbreviated "regex" or "RE")? To answer this, we must return to the overarching question: what is a programming language? This chapter uses two competing notions of language side-by side, so we give them different names: a *programming language* is the broadest notion of a language, as outlined in the chapter "What is a Programming Language". A *string language* is a mathematical set of strings. *Regular expressions are a programming language where every program defines a string language.* These programs can be executed with a string as their input and output a Boolean indicating whether the string belongs to the given string language and, if so, any portions of the string that are of interest to the programmer. Regular expressions are foundational to the implementation of programming languages because processing strings is an essential implementation task. Viewed as a programming language, regular expressions are both a domain-

specific language and a little language: they specialize in the domain of string-processing problems and are typically used as one small piece of a larger application. The name “regular expressions” comes from the Theorist: regular expressions do not attempt to define every possible string language, and the ones they *can* define happen to be known as regular languages. The parts of syntax we define with regular expressions will come to be known as a programming language’s *terminal symbols*, which include symbols such as variables, literals, or operator symbols. What regular expressions cannot do is assemble these basic pieces into the larger structures of a program, such as full expressions, statements, or definitions.

We first define the programming language of regular expressions in general, then define its meaning using string languages, before proceeding to discuss implementations of regular expressions in general-purpose programming languages.

The Programming Language of Regular Expressions

We present every regular expression feature one after another, giving the informal meaning and examples of each. We use r as a mathematical variable to mean “any regular expression.” We

write regular expressions in single quotes to distinguish them from strings.

Character Regular Expressions

Any single character can be written as a regular expression. This specific kind of regular expression is called a character regular expression and we use the mathematical variable c to mean any character regular expression. A character regular expression ' c ' only matches one string, the one-character string " c ". Many punctuation symbols are used for the syntax of other regular expressions, so if we want to use such symbols (e.g. $*$) in a character regular expression, we prefix them with a backslash, e.g., $\backslash*$.

Character regular expressions are of almost no use on their own: no new programming language should need to test individual characters like "A" or "d". Rather, character regular expressions are important because they are the basic building block of other regular expressions.

Sequential composition

Any two regular expressions r_1 and r_2 can be written immediately next to each other with no separator: r_1r_2 ; the combined regular expression r_1r_2 is called the sequential composition of r_1 and r_2 .

To match a string against r_1r_2 , first match the beginning of the string against r_1 and then match the remainder of the string against r_2 .

As with character regular expressions, sequential compositions are only powerful as building blocks for more complex regular expressions. Combining sequential compositions and character regular expressions allows writing a regular expression that matches exactly one string. For example, ‘ab’ matches the string “ab” and ‘hello’ matches the string “hello”.

Choice

For any regular expressions r_1 and r_2 , the choice of r_1 and r_2 is written as ‘ $r_1 | r_2$ ’. The regular expression $r_1 | r_2$ matches a string s if either r_1 or r_2 match s . It is permitted but not required for both expressions to match s .

Combining character regular expressions, sequential compositions, and choices, we can express any finite set of strings as a regular expression. For example, the regular expression ‘just | some | words’ matches only the strings “just”, “some”, and “words”.

Parentheses

We write parentheses to clarify the order of operations in regular expressions, e.g., ‘(a | b)c’ and ‘a | (bc)’ are distinct regular expressions. One should be cautious using parentheses in implementations of regular expressions, because many implementations use them to indicate which sections of a string should be returned to the user as output upon a match.

Repetition

For any regular expression r , we write r^* for the repetition of r . To match a string s against r^* , we repeatedly match the beginning of the string against r and remove it, then repeat. If we can use up the whole string s in this way (i.e., we arrive at the empty string) then the string matches r^* .

The repetition r^* can repeat r any number of times, including zero, so the empty string always matches r^* , no matter what r is. The regular expression r^* also matches every string matched by r , rr , rrr , and so on. For example, if r is ‘ab’ then $(ab)^*$ matches “”, “ab”, “ab”, “abab” and so on.

By combining repetition, choice, and sequential composition, we can start to express nontrivial patterns. For example, the regex ‘(A | B)(a | b | A | B)*’ requires the first letter to be uppercase and

allows all following letters to be either upper or lower-case, and there can be as many letters as we want.

Together, character regular expressions, sequential composition, choice, and repetition make up the core of all regular expressions; this fragment of the language is the basis of most mathematical study. For mathematical convenience, theoretical studies often add one more regular expression which is not needed in most practical applications, the zero regular expression.

Zero

The “zero” regular expression, not to be confused with the character regular expression ‘0’, is written “ with an empty string between the single quotes, and does not match any string, not even the empty string.

Practical usage of regular expressions typically involves many more kinds of regular expressions which, although they do not increase the theoretical power of the language (i.e., they do not define any new string languages that could not be defined before), are essential for convenience. We call these features *definable* or *derivable* features because they can be defined using the core features. In colloquial speech, definable features are often called *syntactic sugar* because they are added

on top of a core language to make it more pleasant. We now define commonly-used definable regular expressions.

Wildcard

The wildcard regular expression, written as a period ‘.’, matches any single character. In principle, the wildcard regular expression can be derived as long as there are a finite number of characters c_i , by writing the choice $(c_1 | c_2 | c_3 | \dots)$ and so on for every character. Such an approach would be untenable even for small sets of characters like the ASCII character set, and would be completely infeasible for multi-lingual representations of text like Unicode, which includes many thousands of characters used to represent dozens of different writing systems. Though the wildcard character is derivable in theory, it is essential in practice.

An example use of a wildcard is parsing a text file that contains a field for a person’s middle initial: the pattern ‘Initial:.’ expects the text file to contain the header ”Initial:“ but allows any character to be filled in for the initial. The wildcard is commonly used in the regular expression ‘.*’ together with repetition to allow an arbitrary string.

To fix this, there is a “wildcard” regex, written as a period, which matches any single character. That is, the expression ‘.’ matches any single character, and if you wish to allow an arbitrary *string*,

you just repeat it: ‘.*’ means any string. For example, the regular expression ‘.*treasure.*’ matches any string that contains the string “treasure”.

Ranges and Sets

Oftentimes, we wish to allow many different character’s but not every character in a regular expression. Such regular expressions are possible, but inconvenient, to express with character regular expressions. Consider the length and complexity of the regular expressions for digits: ‘(0|1|2|3|4|5|6|7|8|9)’ or alphanumeric characters:

‘(0|1|2|3|4|5|6|7|8|9|a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z)’.

We alleviate the challenge of expressing such sets of characters using the definable regular expression features of *ranges* and *sets*, which are written in square brackets. For examples:

- * The regular expression ‘[0-9]’ means “any digit”
- * The regular expression ‘[a-z]’ means “any lowercase Latin letter”
- * The regular expression ‘[A-Z]’ means “any uppercase Latin letter”

To avoid confusion with ranges, the hyphen symbol must be escaped with a backslash ‘[\-]’ to match it within square brackets. We can put more than one range next to each other in square brackets, but in contrast to normal regular expression syntax, placing ranges next to each other does not mean sequential composition, but rather choice, just like ‘|’ syntax in typical regular expression syntax. For example, the regular expression ‘[a-zA-Z]’ means “any Latin letter”. If we write letters next to each other in square brackets without range syntax, this choice of characters is called a set. For example, [aceg] is a set regular expression which matches the four strings “a”, “c”, “e”, and “g”, the same as the more verbose regular expression ‘a|c|e|g’.

Square bracket syntax can also express the concept of negation, i.e., “everything except certain characters”. To do this, put a caret symbol immediately after the opening brace. For example, ‘[^0-9]’ means “any single character except a digit”. To avoid confusion with the negation syntax, if you want to match the literal character ^ in a range, make sure it is not the first character, e.g., you could match the five-function arithmetic operators (+,-,/,*,:) using the range [+*^-^]

Extended Matching

Thus far, this chapter has assumed that the process of matching checks that a regular expression can match (use up) the *entirety* of

a string. This approach is standard in theoretical analysis of regular expressions, but not in implementations. In implementations, it is typical to search for any *substring* of the input that matches a given regular expression. In implementation, but not in theory, one would say the regular expression ‘bcd’ matches the input string ‘abcde’ in the sense that it matches the substring ‘bcd’. For lack of a better term, we refer to the style of matching performed by standard implementations as *extended matching*.

Once the extended matching approach is adopted, several extended regular expressions are needed to recover our ability to match beginnings and ends of strings (typically lines of a file); we call these new regular expressions *begin* and *end*.

The *begin* regular expression is written as the single character ^, which does not conflict with the use of ^ in bracket notation because *begin* never appears in brackets. The *end* regular expression is written as the single character \$. Begin and end are often used together to match an entire line. For any regex r, the regex ‘^r\$’ means “r, but it can only match complete lines”, or in other words, extended matching of ^r\$ behaves identically to standard matching of r.

To demonstrate the importance of including the begin and end symbols, we consider a typical use of regular expressions in software development: searching through a log of debugging

information generated by an application. Suppose the hypothetical program CoolProgram, which builds on the hypothetical library UncoolLibrary, produced the following log messages when run:

```
info: CoolProgram initialized without error
```

```
debug: CoolProgram version: v13
```

```
debug: UncoolLibrary v420.69 loaded without error
```

```
error: UncoolLibrary version cannot be newer than CoolProgram  
version
```

Suppose you search through this log message using the regular expression ‘error’ (feel free to try it out as you read!). This search will match three different lines, because three contain “error” as a substring. On large logs and other large datasets, these extended matches might return so many lines as to be problematic. By adding a begin marker and matching the regular expression ‘^error’ instead, you can remove two spurious matches and match only the final line, as intended.

The Mathematical Meaning of Regular Expressions

This section formally defines the meaning of every core regular expression by defining its string language. In doing so, we only begin to scratch the surface of the theory of regular expressions. Curious readers are referred to existing texts on the foundations of computation for further study.

For any regular expression r , we write $L(r)$ for the (string) language of r , which is always a set of strings. $L(r)$ is defined recursively by the equations:

- * $L(\epsilon) = \{\}$
- * $L(c) = \{c\}$
- * $L(r_1 r_2) = L(r_1) \circ L(r_2) = \{ s_1 s_2 \mid s_1 \text{ in } L(r_1) \text{ and } s_2 \text{ in } L(r_2) \}$
- * $L(r_1 | r_2) = L(r_1) \cup L(r_2) = \{ s \mid s \text{ in } L(r_1) \text{ or } s \text{ in } L(r_2) \}$
- * $L(r^*) = L(r)^* = \{ s_1..s_k \mid k \text{ in } \mathbb{N}, s_i \text{ in } L(r) \text{ for every } i \text{ from } 1 \text{ to } k \}$

To determine the language of a given regular expression r , we apply the above rules repeatedly. For example, we compute the language of ' $(a|b)^*$:

$L('ab)^* = \{s_1..s_k \mid k \in \mathbb{N}, s_i \in L('a|b') \text{ for every } i \text{ from 1 to } k\} =$

$L('a|b)^* = \{s_1..s_k \mid k \in \mathbb{N}, s_i \in L('a') \text{ or } s_i \in L('b') \text{ for every } i \text{ from 1 to } k\} = \{s_1..s_k \mid k \in \mathbb{N}, s_i \in \{"a", "b"\} \text{ for every } i \text{ from 1 to } k\}$, at which point every language has been fully expanded, and we have fully described the mathematical meaning of the regular expression '(ab)*'. Mathematically defining regular expressions in this way has multiple uses for the Theorist, such as determining whether two regular expressions are equivalent, determining whether a regular expression successfully implements a given specification, or assessing the correctness of tools that implement regular expressions.

The Implementation of Regular Expressions

We briefly discuss existing implementations of regular expressions in several languages. This discussion is kept to avoid redundancy with implementation discussion in the Parsing Expression Grammar chapter, which discusses a more expressive implementation technology.

Rust

In Rust, regular expressions are implemented by the `regex` crate, whose documentation is located

at <https://docs.rs/regex/latest/regex/>. To use this crate, add it to your Cargo.toml file and then write

```
use regex::Regex;
```

at the top of your file. From the documentation, an example regular expression for dates can be written:

```
let re = Regex::new(r"^\d{4}-\d{2}-\d{2}$").unwrap();
```

where `Regex::new` creates a new regular expression, the `r` at the beginning of a string literal indicates raw string syntax, the syntax `d{k}` means a sequence of `k` digits, and `unwrap()` implements error handling. Once created, the regular expression can be checked to match a given string using the `is_match` method;



Rust supports capture groups, which are typically accessed using iterators. From the documentation:

```
let re = Regex::new(r"(d{4})-(d{2})-(d{2})").unwrap();
```

```
let text = "2012-03-14, 2013-01-01 and 2014-07-05";
```

```
for cap in recaptures_iter(text) {  
  
    println!("Month: {} Day: {} Year: {}", &cap[2], &cap[3],  
    &cap[1]);  
  
}
```

where `captures_iter(text)` matches the given text against the given string and provides an iterator over the captured substrings.

Scala

To make use of the built-in regular expression library, add the following line to the beginning of your file:

```
import scala.util.matching.Regex
```

This adds a method named “r” to all strings, which allows converting them into regular expressions. Traditionally, the raw string syntax is used when creating regular expressions, to ensure correct handling of escape characters. Thus, to interpret the string “myRegularExpression” as a regular expression, we

write `raw"myRegularExpression".r` The Scala syntax for regular expressions is the same as in

Java: <https://docs.oracle.com/javase/8/docs/api/java/util/regex/Pattern.html>

Note that in this notation, parentheses indicate *capturing groups*, which indicate which substrings should be extracted from the matched string as outputs of the matching algorithm.

In Scala, regular expression matching is integrated with the existing syntax for pattern-matching, as demonstrated by the following example from the Scala documentation at <https://www.scala-lang.org/api/2.13.4/scala/util/matching/Regex.html>:

```
val date = raw"(d{4})-(d{2})-(d{2})".r
```

```
"2004-01-20" match {
```

```
    case date(year, month, day) => s"$year was a good year for  
    PIs."
```

```
}
```

Which creates a regular expression for dates, matches it against a string, and uses the extracted substrings in the subsequent code. If you wish to check whether a string matches a regex, but do not use the capture groups, then you can use a special “wildcard

argument” pattern match, distinct from but sharing a common theme with the wildcard regex:

```
//also from the above Scala documentation
```

```
val date = raw"(d{4})-(d{2})-(d{2})".r
```

```
"2004-01-20" match {
```

```
    case date(year, month, day) => s"$year was a good year for  
    PIs."
```

```
}
```

Classroom Activities

1. Write example regular expressions and strings on the board. Have students guess whether the regular expression matches the string, then walk through an informal matching algorithm on these examples.
2. Write an incorrect regular expression and ask students to find an example string on which it breaks.
3. Demonstrate using regular expressions to search through large bodies of text
4. Solve a code golf exercise and present the process of solving it

5. Work through examples of how to derive definitions for new regular expressions from existing ones, such as deriving non-zero repetition (r^+) and optionality ($r?$) from the core operators.
6. Discuss: When is it better to solve a string-matching problem with regexes vs some other implementation strategy? My regex for decimal numbers is complex and hard to read. Is this “worth it” if it is much more concise than other implementation approaches?

Exercises

1. Solve regex golf puzzles
2. Define a type in your programming language of choice, whose values are all the core regular expressions. Usually, this will be defined inductively.
3. Define regular expressions for the following string languages:
 - a. The language of single digits (0 through 9)
 - b. The language of natural numbers (include 0, but do not allow leading 0s on positive numbers)
 - c. The language of integers, which allow leading minus signs on natural numbers, but do not allow -0
 - d. Floating-point numbers. Your syntax should not allow leading zeros on the integer part, should not allow a

leading minus sign on numbers whose value is zero, and should only allow a decimal point when the decimal point is followed by at least one digit. It is permissible for the digits following the decimal point to be all zeros.

- e. Variable names that follow the following rules: a variable name must start with a letter. Afterward, it can contain any combination of letters, digits, underscores, hyphens, and apostrophes. No other letters are allowed, and uppercase and lowercase letters are both allowed.
- f. Dates in the mm/dd/yyyy format which is popular in the United States. Leading zeros should be required for months and days 0-9.
- g. Dates in the yyyy-mm-dd format standardized by the ISO. Leading zeros should be required for months and days 0-9.
- h. Strings that follow the following password rules: at least one lowercase character, at least one uppercase character, at least number. Assume for simplicity that the password contains only letters and numbers.
- i. Strings that follow these password rules in addition to those above: at least one special character, which must belong to the set {@#\$%^&*}, length between 10 and 20. For this problem only, it may help to assume you have access to the “intersection” regular expression.
- j. 10-digit phone numbers as commonly used in the United States, in “(ddd) ddd-dddd” format where each d is a digit

- k. A hexadecimal number, i.e., a series of digits where each digit is 0,1,2,3,4,5,6,7,8,9,a,b,c,d,e, or f. The letters can be upper or lower-case. Hexadecimal constants are often prefixed with “0x” in programming languages but this problem does not ask for the prefix
- l. An IPv6 address of form $y:y:y:y:y:y:y:y$ where y means any four-digit string of hexadecimal digits
- m. A 48-bit MAC address as defined by the IEEE 802 standard, of form $b:b:b:b:b:b$ where b means any two-digit string of hexadecimal digits
4. Define regular expressions for the following string languages, which are motivated by theory and math:
- The language of all polynomial expressions of form cx^k where c is any decimal number, k is any integer, and x is any single-letter variable name
 - The language of all polynomial expressions of form $c_1x_1^k_1 + c_2x_2^k_2$ where c_1 and c_2 are decimal numbers, k_1 and k_2 are integers, and x_1 and x_2 are single-letter variables.
 - The set of all strings of the form $a^{\{2n\}}b^{\{2m\}}$, meaning any even number of a's followed by any even number of b's
5. A Practitioner must often design regular expressions to solve an application-specific programmer, and the Social Scientist might observe that different Practitioners living in different

social contexts may arrive at different answers. The following problems do not have only one string language as their solution. Design a solution, discuss why it is appropriate for your context, and discuss how someone else might solve it different depending on their context:

- a. Names as they would appear on name tags at a major professional event such as an open house or a conference. Consider the following questions: What characters can appear in names? Should names start with titles such as Dr.? How many names (e.g. given names, middle names, hyphenated) can someone have? Can names be hyphenated? Which ones? Should pronouns be listed after names? Which ones? Should that be required?
- b. Phone numbers as they would be collected in emergency contact information for a large group gathering or trip. Consider the following questions: What countries' phone numbers do you need to support? How do you handle diversity of formats across different countries? How do you handle phone numbers with extensions? How do you handle data where some phone numbers may include country codes and others might not?
- c. Email addresses as they would be used in a sign up page for an account on a website. Consider the following questions: What characters are allowed in an email address? What are the required parts of an email address? What domain names are allowed? Are there

any domain names which, although they are technically allowed, you wish to prohibit from signing up, e.g., due to concerns of security-related attacks on a website?

- d. File paths as you would enter them in a command-line. Consider the following questions: Should paths be relative, absolute, or both? Should they use drive names as in Windows or omit them as in Linux and Mac OS? How long can paths be, and is it worth checking length? What characters can appear in a path? Should file extensions be required and, if so, which ones? Some of these answers will depend on which operating system you use and which filesystem your computer uses.
- e. Password requirements as one would use in the account signup page of a web application. Consider the following question: What have security and human-computer interactions recommended about the design of password requirements?
- f. Postal addresses as they would processed on an e-commerce website. Consider the following questions: How do you deal with shipping addresses that ship to other countries? How do you deal with shipping addresses within your own country that are written in the format of another country? Which parts of an address are optional and which are required? Which fields should have a restricted, finite set of allowed values for error-checking? How do you handle variable-

length postal codes such as the United States system of ZIP and ZIP+4 codes?

- 6.** Which of the following languages are regular?
 - a.** The language of all polynomial expressions, which can include arbitrarily nested addition, multiplication, and parentheses
 - b.** The language of polynomial expressions of the form $c_1x_1^{k_1} + c_2x_2^{k_2}$ specifically
 - c.** The programming language you are using to implement the exercises from this book
 - d.** The set of all sentences that follow English word order (V2 word order) using words defined in the latest edition of the Oxford English Dictionary.
 - e.** The set of all English sentences that you personally have spoken so far in your life (tricky!)
 - f.** The set of well-matched strings of parentheses
 - g.** The set of strings that start with capital A, are immediately followed by any number of lowercase a's, and end with the characters h and ! in that order.
 - h.** The set of valid dates in mm/dd/yyyy format
 - i.** The set of valid IPv6 addresses
- 7.** Using that datatype, implement functions that transform the “derivable” regular expressions into “core” regular expressions:

- a. Nonzero repetition r^+
 - b. Optional expression $r^?$
 - c. Range expression $[c_1-c_2]$
 - d. Set expression $[c_1c_2\dots c_N]$
 - e. Wildcard expression . (when the alphabet Sigma is known)
8. For each of the following pairs of regular expressions, determine whether (i) have the same language, (ii) the first language is a strict subset of the second, (iii) the second language is a strict subset of the first, or (iv) each language contains at least one string that the other does not. If your answer is (ii) or (iii) provide one example string which belongs to the larger language but not the smaller. If your answer is (iv), give two example strings, one that is exclusive to each language
- a. '(a | b)(c | d)' and '(ac) | (bd)'
 - b. 'a*a' and 'aa*'
 - c. 'ab*' and 'a*b*'
 - d. 'a*b*' and '(ab)*'
 - e. 'acab' and '(ac)a(b)'
 - f. 'hey' and 'hey|hello'
 - g. 'a*' and 'aa*'

- h. ‘a^{*}’ and ‘(aa)^{*}’
 - i. ‘b^{*}’ and ‘b^{**}’
9. Implement a program that recognizes a specific regular expression, without using a regular expression library. Now compare it to an implementation that uses the library. How does the performance differ? How does the implementation approach differ?
10. Implement a program that recognizes an arbitrary regular expression, by recursion on the regular expression type
11. Extend your recursive function to support a new intersection regular expression $r_1 \cap r_2$ whose language is the intersection of the languages of r_1 and r_2 . Note: This problem is a well-known problem for teaching continuations in functional programming courses. It is encouraged to learn continuations first. This problem may be challenging without guidance
12. Project for the Social Scientist: Build a set (corpus) of modern programs that use regular expressions. You could do this by picking some common programming language such as C or Java, learning what regular expression libraries are used in those languages, and searching for programs that use those libraries on websites such as GitHub. Take detailed notes of how you chose your corpus of programs. Next, formulate a question about the corpus and search through the corpus to answer it. Example question: What are the most common applications of regular expressions today? How often are

programming languages the application, as opposed to other string-processing tasks?

- 13.** Project for the Implementer and Theorist: Learn about different kinds of state machines that can implement regular expressions, such deterministic finite automata (DFAs) and nondeterministic finite automata (NFAs). A well-known article purports that NFAs provided better performance in its author's use cases than DFAs did, at the time of its writing. This observation is of theoretical interest because the theory of algorithms would suggest that DFAs are faster in theory. Reproduce the tests from that article: <https://swtch.com/rsc/regexp/regexp1.html> and determine which one is faster today.
- 14.** Project for the Humanist: Write an essay exploring the impact of character set choices on the inclusivity of regular expression-based software toward users whose native writing system goes beyond the Latin alphabet. One approach is to explore the divergent experiences of writing software for the Latin alphabet, which has a standard and well-published notation, vs. writing universal code that supports all major writing systems, which often relies on extended regular expression features that are specific to a given software library and are not well-advertised. Another approach is a cultural study on what language means to language designers: for example, some Chinese language designers have used Chinese characters for this reason, with at least one focusing on Classical Chinese in particular. A third

approach is to engage with a linguistic angle focused on casing, how many written languages do not feature case distinctions, and how case distinctions arise in regular expressions. For further discussions of different notations and their accessibility to different groups, see the chapter on Disability.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter5

Chapter 5

Context-Free Grammars

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * Whereas regular expressions define simple non-recursive string languages, context-free grammars can define recursive string languages such as the syntaxes of programming languages
- * Context-free grammars are the basis of many tools for implementing parsers in programming languages
- * Context-free grammars are the starting point for formal grammars used to model human languages
- * To show that a string belongs to the language of a context-free grammar, we write a derivation

- * Some context-free grammars are ambiguous, meaning that the same string might be parsed in two different ways. Though not all forms of ambiguity are fixable, the most common forms, which involve precedence or associativity, are fixable.
- * Context-free grammars have been used as the basis of a programming language for procedural art generation, called Context Free Art.

Context-free grammars (CFGs), like regular expressions, are a domain-specific language for defining string languages, a little language used in the implementation of parsing the syntax of a programming language. They are a language for defining languages.

Context-free grammars are motivated by the limitations of regular expressions, which can only represent a simple class of languages called *regular languages*. To a first approximation, regular expressions can only parse flat structures, meaning that they can parse repeated structures but cannot parse recursive structures. Consider the driving example of parsing polynomial expressions: a regular expression can parse fixed structures such as ‘x * y’ or ‘x + y’. Using repetition, a regular expression can even parse repeated patterns such as “a + b + c + d + e”. However, if we allow polynomial expressions to contain parentheses (for example: “((a*b)+c)*(d-z)”), then regular expressions are not capable of

detecting whether the parentheses are well-matched. In fact, parenthesis-matching is the classic example of a problem that cannot be solved with regular expressions:

Theorem: Let S be the set of strings of well-matched parentheses. There does not exist any regular expression r such that $L(r) = S$.

Proof: Consult standard texts on the foundations of computation. The proof relies on a theorem that every regular language can be implemented as a state machine with finitely many states, and a lemma known as the Pumping Lemma which expresses the limited power of such state machines.

Context-free grammars will allow us to express the recursive structure of programming language syntax. That recursive structure is built up from smaller pieces of syntax, called terminal symbols. In this chapter, we will continue to use regular expressions for terminal symbols and use Context-free grammars only for building them up into more complex (non-terminal) programs. Though context-free grammars can parse everything that regular expressions can parse, we use both approaches together because it more accurately reflects a common approach taken in implementation. In our polynomial example, variable names, numbers, and mathematical operators are each recognized by regular expressions, then combined into polynomial expressions using CFGs.

Defining Context-Free Grammars

The formal mathematical definition of a context-free grammar (variable name: G) has four parts:

- * The set of *variable symbols* (name: V), each of which is a string naming one piece of the syntax that can be built up from smaller pieces,
- * The set of *terminal symbols* (name: Σ , pronounced “sigma”), each of which is a string naming a basic building block that is not built up from smaller pieces,
- * The set of *production rules* (name: P) which define how every variable symbol can be built up from smaller pieces , and
- * The start symbol (name: S), a string indicating which variable symbol we wish to match strings against (i.e., which variable symbol represents a complete program

Depending on your needs, context-free grammars can either be defined in a fully formal way or a semi-formal way. We present both definitions.

Formal Definition

A context-free grammar is a four-element tuple (V, Σ, P, S) such that:

- * $V \cup \Sigma \subseteq \text{string}$ (all symbols are strings)
- * $V \cap \Sigma = \{\}$ (variable and terminal symbols are different)
- * $S \in V$ (the start symbol is a variable symbol)
- * $P \subseteq V \times (V \cup \Sigma)^*$ (every production produces one variable symbol from a sequence of symbols, each variable or terminal)

Notably, the formal definition of a context-free grammar does not specify the regular expressions for each terminal symbol. One could define a *complete grammar* for a programming language as a five-element tuple (V, Σ, P, S, R) where $R : \Sigma \rightarrow \text{regex}$ defines a regular expression $R(s)$ for every terminal symbol “ s ”.

Semi-Formal Definition

When writing out a context-free grammar, we typically write it in the following semi-formal notation, from which the formal four-tuple is implied. A context-free grammar in semi-formal notation is written as a series of production rules, where every production rule is written with the produced variable symbol on the left, then a right arrow, then a space-separated sequence of the symbols from which it is produced. As an example, we provide a grammar for polynomials. In the following grammar, S stands for “start symbol,” N stands for “numeric literal,” I stands for “identifier for a variable of a polynomial,” LP and RP stand respectively for “left

parenthesis” and “right parenthesis”, and O stands for “any mathematical operator”.

$S \rightarrow N$

$S \rightarrow I$

$S \rightarrow LP \ S \ RP$

$S \rightarrow S \ O \ S$

To recover a formal four-element tuple from this notation, we follow these conventions:

- * There is always a variable symbol named S, which is always the start symbol
- * Every variable symbol has at least one rule, i.e., V is defined as the set of all symbols that appear on the left hand sides of arrows
- * Thus Σ is the set of all symbols that appear on the right hand sides, but not left hand sides of arrows
- * Every production rule $v \rightarrow s_1 \dots s_N$ corresponds to an element of P, which would formally be written $(v, s_1 \dots s_N)$

Test your understanding: try to convert the polynomial example to its tuple form before checking the solution

Solution: In fully formal notation, the polynomial grammar is written $(\{S\}, \{N, I, LP, RP, O\}, \{(S, N), (S, I), (S, LP S RP), (S, S O S)\}, S)$

On the first reading, you may find the context-free grammar for polynomials cryptic because it only describes the recursive structure of a polynomial and tells us nothing at all about the meanings of the terminal symbols I, N, O, LP, RP. To finish creating a complete grammar, we define regular expressions for every terminal symbol. For the needs of this chapter, the following definitions suffice:

- * I = '[a-zA-Z]+' meaning any nonempty sequence of letters
- * N = '0 | -?[1-9][0-9]*' meaning any integer
- * O = '\-+*' meaning the symbols for subtraction, addition, or multiplication
- * LP = '\'(' meaning a left parenthesis, and
- * RP = '\)' meaning a right parenthesis.

This completes our first fully formal definition of the syntax for a recursive language. The effort we have invested in careful definition will pay off, as the same formal structure, context-free grammars in combination with regular expressions, is sufficient for defining complex, production-grade programming languages.

The difference between the language of polynomials and a full programming language syntax is merely a matter of scale.

Alternative Notations

Context-free grammars are such a pervasive tool for describing syntax that multiple equivalent notations have evolved for describing production rules. In this chapter, we consistently write each rule separately, but a common alternate notation is to group the rules by the symbol they produce, combining the rules into one line and separating the right-hand-sides with a vertical bar, such as in:

$$S \rightarrow N \mid I \mid LP\ S\ RP \mid S\ O\ S$$

Furthermore, the symbol $::=$ is often used in place of the symbol \rightarrow , particular in works that wish to use the symbol \rightarrow for some other purpose:

$$S ::= N \mid I \mid LP\ S\ RP \mid S\ O\ S$$

Though we do not use these alternative notations, you should be familiar with them when reading other works.

From Grammar to Language: Derivations

To formally understand the meaning of a grammar, we can define its (string) language, the set of all strings which match (are accepted by) the grammar. This is the same basic approach as we took to understand regular expressions, but defining the languages of context-free grammars (called context-free languages) will require an additional tool: derivations. A derivation is a sequence of replacement steps which explain how a string is produced from a context-free grammar. As before, we give both formal and semi-formal definitions.

Formal Definition

A derivation step is a nested tuple $((B_1, b_2, B_3), (E_1, E_2, E_3))$. The first element $B = (B_1, b_2, B_3)$ represents the state of the derivation at the beginning of the step and the second element (E_1, E_2, E_3) represents the state of the derivation at the end of the step. The following conditions must hold:

- * $B_1, B_3, E_1, E_2, E_3 \subseteq (V \cup \Sigma)^*$, i.e., they are sequences of symbols
- * $b_2 \in V$, i.e., it is a variable symbol
- * $(b_2, E_2) \in P$, i.e., some production produces b_2 from E_2

This definition, though cryptic at first, indicates that in each derivation step we divide our current string $B_1b_2B_3$ into a prefix B_1 , a symbol b_2 we wish to derive, and a suffix B_3 . The prefix and suffix are left untouched, and we replace b_2 according to any of its production rules.

A derivation is a sequence of derivation steps $((B,E), (B',E'), \dots)$ such that:

- * For the initial step $((B_1,b_2,B_3),(E_1,e_2,E_3))$ we have $b_2 = S$ and B_1, B_3 empty, i.e., the derivation starts with the start symbol
- * For any two adjacent derivation steps (B,E) and (B',E') , we have $E = B'$ when interpreted as strings, i.e., each step begins from the ending string of the previous step

We define the language of a context-free grammar G in terms of its derivations: $L(G) = \{ E_1E_2E_3 \text{ such that there exists some derivation for } G \text{ which ends in } (E_1,E_2,E_3)\}$

That is, to produce the language of a context-free grammar, we produce all its derivations, then take the strings that were derived.

The strings in the language of a context-free grammar only use the symbols from that grammar. In the polynomial example, the language contains strings like “N O N” but not strings like “1 + 2”.

To capture the second notion of language, we can consider a

complete grammar (G, R) where R contains regular expressions for every terminal symbol. Interpret $L(G)$ as a set of strings over symbols (i.e., $\{ s_1 \dots s_N \}$), then we define $L(G, R) = \{ str_1 \dots str_N \mid \exists s_1 \dots s_N \in L(G) \text{ such that } str_i \in L(R(s_i)) \text{ for all } 1 \leq i \leq N \}$. That is, a string matches a complete grammar (G, R) if it matches *any* sequence of terminals in the context-free language $L(G)$, as defined by the regular expressions R .

Semi-Formal Definition

The semi-formal notation for derivations consists of one line per derivation step, with the beginning and end separated by an arrow. On the left, the symbol to be rewritten is and its replacement are written **in bold**; if you wish to type the semiformal notation in plaintext, then you can surround the symbol with underscores instead. An arrow with an asterisk $A \rightarrow^* B$ indicates that B derives from A in any number of steps, not necessarily one step.

The following example derives the string “N O N O I” from the start symbol S of the polynomial grammar:

$S \rightarrow S O S$

$S O S \rightarrow S O S O S$

S O S O S → N O S O S

N O S O S → N O N O S

N O N O S → N O N O I

The resulting string “N O N O I” consists entirely of terminal symbols and is thus ready to be matched against the regular expressions for those symbols. To derive a final string from these terminal symbols, we can match the symbols one at a time:

N O N O I → 1 O N O I

1 O N O I → 1 + N O I

1 + N O I → 1 + 1 O I

1 + 1 O I → 1 + 1 * I

1 + 1 * I → 1 + 1 * x

Thus the derivation has shown that the string “1 + 1 * x” belongs to the language of polynomials.

Check your understanding: write a derivation that “y * 2 - x” belongs to the language of polynomials, then check it against the solution below. First derive a sequence of terminal symbols, then

derive the final string from the nonterminal symbols. Note that there is more than one correct solution.

Solution:

We first derive the string “I O N O I” from the start symbol S

S → S O S

S O S → S O S O S

S O S O S → I O S O S

I O S O S → I O N O S

I O N O S → I O N O I

and then derive the final string “y * 2 - x”:

I O N O I → y O N O I

y O N O I → y * N O I

y * N O I → y * 2 O I

y * 2 O I → y * 2 - I

$y^* 2 - I \rightarrow y^* 2 - x$

which completes the derivation of the string “y * 2 - x”

Order of Operations: Ambiguity

Thus far, we have developed a grammar which recognizes polynomial expressions, but we have made no claims about the order of operations within a polynomial. That is, if we parse a polynomial that was written without parentheses, we do not know where the parentheses belong. Without addressing order of operations, we cannot claim to have implemented the syntax of polynomials because they are mathematical expressions, and mathematical expressions follow a standardized order of operations as represented in the acronym PEMDAS:

- * Parentheses (e) have the highest precedence
- * Exponents e^e have the next-highest precedence. Parentheses are placed right-to-left: e^e^e is parenthesized $e^e(e^e)$
- * Multiplication e^*e and Division e/e have the next-highest precedence, and the same precedence as each other. Parentheses go left-to-right: e/e^e is $(e/e)^e$.
- * Addition and Subtraction $e+e$ and $e-e$ have the lowest precedence, and the same precedence as each other. Parentheses go left-to-right: $e-e+e$ is $(e-e)+e$

If we take an input string and its derivation, then add parentheses to the input string according to the structure of the derivation, we call the resulting parenthesized string a *parse* of the input string. A parse is often rendered as a *parse tree*, where each operation corresponds to a node of the tree and its children are its operands. To convert a derivation into a parenthesized string, consider each step $((B_1, b_2, b_3), (E_1, E_2, E_3))$ which replaces some $b_2 \rightarrow E_2$. The example derivation of “ $1 + 1 + x$ ” can be parenthesized as follows:

$$S \rightarrow (S \ O \ S)$$

$$(S \ O \ S) \rightarrow ((S \ O \ S) \ O \ S)$$

$$((S \ O \ S) \ O \ S) \rightarrow^* ((1 + 1) + x)$$

Check your understanding: Parenthesize the string “8-4-2” based on its example derivation from this chapter. What string do you get?

Solution: “ $((8-4)-2)$ ” for the same reason as in the example “ $((1 + 1) + x)$ ”

In both of the example derivations, the steps were applied in a string left-to-right order, which led to left-to-right parentheses. We got lucky in those examples because they each contained a single

operator, for which left-to-right parentheses are correct. However, problems arise when we mix operators or use repeatedly use a right-to-left operator. If we apply the same derivation structure to the string “ $1+1*2$ ” we would get “ $((1+1)*2)$ ” which is incorrect; the correct parentheses are “ $(1+(1*2))$ ”. These expressions have entirely different values: 4 and 3. Our polynomial grammar does not contain exponentials, but a similar issue would arise when parsing the expression “ 3^3^3 ” which would parse as $((3^3)^3)$ “ when the correct result is ” $(3^{(3^3)})$ “, a much larger number.

Check your understanding: Can you use the polynomial grammar to derive the correct parenthesized string “ $(1+(1*2))$ ” from the start symbol S?

Solution: Yes, by the following derivation:

$$S \rightarrow (S \ O \ S)$$

$$(S \ O \ S) \rightarrow (S \ O \ (S \ O \ S))$$

$$(S \ O \ (S \ O \ S)) \rightarrow^* (1+(1*2))$$

The fact that we can derive the correct parenthesization should not be cause for celebration, because our grammar still *allows* the incorrect parenthesization. If the job of a grammar is to recognize the precise structure of a parsed string, our grammar has failed. It

has failed because it is an ambiguous grammar and carries ambiguous meaning.

Definition: A grammar G is ambiguous if there exist two derivations D_1 and D_2 for G for the same input string which produce two different parses, i.e., two different parenthesized strings or parse trees.

The two derivations of string “ $1+1*2$ ” which respectively result in parentheses “ $((1+1)*2)$ ” and “ $(1+(1*2))$ ” constitute a proof that the grammar is ambiguous.

Ambiguity comes in several forms. For certain context-free languages, their *only* context-free grammars are ambiguous, thus not all ambiguity can be removed in every case. However, such grammars are rare in practice. As demonstrated through our examples, by far the most common forms of ambiguity are ambiguous precedence (mixing two operators) and ambiguous associativity (using the same operator twice). Precedence and associativity can both be fixed in systematic ways, which we now present.

Fixing Ambiguity

We consider precedence first, then associativity

Fixing precedence ambiguity

Recall that in parsing “ $1+1*2$ ”, the example grammar does not know whether + vs * takes priority, i.e. it cannot determine which rule should be specified first. W

We wish to require, through our grammar, that * has a higher precedence level than +, so that the *only* derivation is the correct one. To achieve this, we must reorganize our grammar and add new variable systems, but we do so in a systematic way. In the new grammar, every precedence level corresponds to its own variable symbol. The lowest-precedence operator is assigned to the start symbol and any higher levels of precedence are assigned increasing indexed names such as E1 for the next level of precedence in expressions. We construct the grammar according to the following requirements:

- * Each variable symbol has rules for the operators at that precedence level. In these rules, the operands are at the same precedence level,
- * Each variable symbol except the highest-precedence has a rule which produces it directly from a symbol at one level higher
- * Terminals are at the highest precedence level

- * Parentheses are at the highest precedence level, and the expression inside them is at the lowest precedence level

Check your understanding: Try to rewrite the grammar for polynomials before checking the solution below.

Solution:

$$S \rightarrow S + S$$

$$S \rightarrow S - S$$

$$S \rightarrow E1$$

$$E1 \rightarrow E1 * E1$$

$$E1 \rightarrow I$$

$$E1 \rightarrow N$$

$$E1 \rightarrow LP S RP$$

This grammar requires that the rules for symbol S are tried “at the outside” and rules for symbol E1 are tried “at the inside”, thus + takes precedence.

Fixing associativity ambiguity

The second source of ambiguity is when an operator appears twice, e.g., 8-4-2. Then we do not know which instance of the operator we should prioritize. The order in which we prioritize these instances is called its associativity.

Definitions: An operator O is *left-associative* if “e1 O e2 O e3” is parsed the same as “(e1 O e2) O e3”. An operator O is *right-associative* if “e1 O e2 O e3” is parsed the same as “e1 O (e2 O e3)”. An operator O is *associative* if both “(e1 O e2) O e3” and “e1 O (e2 O e3)” have the same meaning. Even when an operator is associative, we typically parse in a left-associative or right-associative way, to remove ambiguity.

Associativity is important precisely because some operators, such as subtraction, are not associative. For example, it is essential to determine whether “8-4-2” parses as “(8-4)-2” or “8-(4-2)” because they have different values: 2 and 6 respectively. By convention, the correct value is 2: subtraction is *left-associative*.

We enforce associativity at the same time as precedence by adding one more requirement in addition to those for precedence:

- * If an operator O is left-associative, then only its left argument is at the same precedence level and its right argument

increases a level (vice versa for right-associative operators)

This requirement enforces that at any derivation step, the present operator can only reappear on the left side, because the right side must increase by a precedence level. In doing so, we make our polynomial grammar fully unambiguous. Note that this requirement, though seemingly simple, can require adding an extra level of precedence with an extra variable symbol. In the polynomial grammar, for example, terminals and multiplication belong to different precedence levels.

Check your understanding: Attempt to add associativity to the polynomial grammar yourself before checking the solution.

Solution: The rules for multiplication, addition, and subtraction are changed; all three are made left-associative. A new variable symbol E2 is introduced, which represents that terminals are higher precedence than multiplication.

$S \rightarrow S+E1$

$S \rightarrow S-E1$

$S \rightarrow E1$

$E1 \rightarrow E1 * E2$

E1 → E2

E2 → I

E2 → N

E2 → LP S RP

This completes the discussion of solutions to ambiguity. The approaches discussed thus far are sufficient to build complex grammars for complex programming languages.

Additional Examples

For additional practice with context-free grammars, we present two example grammars, one from linguistics and one from programming languages.

Linguistics Example

Context-free grammars, though widely used in computer science, actually originate in linguistics, where they are used to model the grammar of natural languages. Though context-free grammars cannot model all human language, they can model much of it. The methodology for building a context-free grammar typically varies between linguistics and computer science. Because natural

language evolves in a more fluid way than programming language notations do, a linguist would take special care to model grammar in a *descriptive* way, working to capture the language of complexity as it is actually spoken, whereas a programming language designer gets to *prescribe* which programs are valid syntax. Because this book focuses on programming languages, we do not undertake the linguistics work of accurately describing all of English. Instead, we use a small fragment of English grammar as an example.

A basic declarational English sentence has a subject and a predicate, represented as a noun phrase (NP) and verb phrase (VP). Noun phrases and verb phrases then incorporate additional parts of speech such as determiners (D), adjectives (Adj) and adverbs (Adv). A determiner adds specificity to a noun, e.g. the amount of it or which one it is. Articles like “a”, “an,” and “the” are also determiners. In this example, we treat N, V, D, Adj, and Adv as terminal symbols.

$S \rightarrow NP\ VP$

$NP \rightarrow D\ N$

$NP \rightarrow N$

$NP \rightarrow Adj\ NP$

VP → V

VP → VP NP

VP → VP Adv

Due to the complexity of human language, the terminal symbols N, V, D, Adj, and Adv each contain a huge number of strings representing different words in each part of speech. Instead of defining these terminals arbitrary, we consider some classic sentences from linguistics and choose the terminals to suffice for match those sentences:

- * Fruit flies like a banana
- * I made her duck
- * Buffalo buffalo buffalo Buffalo buffalo
- * Colorless green ideas sleep furiously

Check your understanding: sort all the words from these example sentences into nouns, verbs, determiners, adjectives, and adverbs. Many of these words serve multiple parts of speech, and you should choose your answer based on how they would be used to construct a valid sentence according to our example grammar. You may put the same word in multiple parts of speech. Some of your

choices may be unintuitive. There are also multiple plausible solutions, of which we give only one.

Solution:

- * Nouns: “flies”, “banana”, “I”, “duck”, “buffalo”, “ideas”, “buffalo”
- * Verbs: “like”, “made”, “buffalo”, “sleep”
- * Determiners: “a”, “her”
- * Adjectives: “Fruit”, “Buffalo”, “Colorless”, “green”
- * Adverbs: “furiously”

Grammars for natural languages are an excellent source of practice for deriving string from a context-free grammar.

Check your understanding: Derive the sentence “Colorless green ideas sleep furiously” from the start symbol S.

Solution:

S → NP VP

NP VP → Adj NP VP

Adj NP VP → Adj Adj NP VP

Adj Adj NP VP → Adj Adj N VP

Adj Adj N VP → Adj Adj N VP Adv

Adj Adj N VP Adv → Adj Adj N V Adv

Adj Adj N V Adv →* Colorless green ideas sleep furiously

In reflecting on the syntax of natural language, we can also enrich our thinking on the syntax of programming languages. The example sentence “Colorless green ideas sleep furiously” is meant to demonstrate that syntactically valid sentences do not always carry semantic meaning. The same idea holds true in programming languages, as we will discuss in the chapter on Type Systems. In modeling a context free grammar, a linguist tries to strike a difficult balance between “Is every string accepted by the grammar syntactically valid in the language?” and “Does the grammar accept every valid string from the language?”. The same tension holds in designing a grammar for a programming language: a designer enough programs to meet the needs of their language, but no more. On other issues, linguistics and programming languages face the same problem but must employ different solutions. Sentences like “I made her duck” and “Time flies like an arrow; fruit flies like a banana” indicate ambiguity in English grammar, just as we experienced ambiguity in a programming language. Because a programming language is

constructed, ambiguity can typically be removed. Because linguistics is descriptive, the linguist has little choice but to accept and document ambiguity.

Functional Programming Language Example

Let's explore how CFGs are used to represent the grammars of programming languages. At the minimum, there must be one variable symbol for each major component of the syntax (expressions, definitions, and, if the language has them, statements). Depending on the complexity of the syntax and the preferences of the designer, the syntax could be divided into further variable symbols. For example, we will use a separate variable symbol for values, but it is equally viable to combine the symbols for expressions and values. Our example language is purely functional, so it will not have statements. Before we attempt to write the context-free grammar for our language, it is important to decide what features the syntax will have. In this case, our main variable symbols will be:

- * The start symbol S which represents a whole program. For the purposes of this example, a whole program is a non-empty series of definitions
- * The expression symbol E, which includes values, variables, let-expressions, arithmetic operations +, -, and *, function calls, if-then-else conditionals, and parentheses. Let

expressions start with “let”, then a single definition, then “in”, then a body expression. Conditionals start with “if”, then a condition, then a branch expression in curly braces, then the keyword “else”, then the other branch expression in curly braces.

- * The value symbol V for values only
- * The definition symbol D, which includes variable and function definitions. Variable definitions have a variable name followed by “=” then its definition. Function definitions have a function name, followed in parentheses by a comma-separated list of argument names, then “=”, then a body expression.

Because arithmetic expressions have two precedence levels, we know we will use at least three variable symbols. Prefix operators like “if” should be an even lower precedence than addition and subtraction, you should plan to use four variable symbols for expressions. Also, because function arguments are a list, they will need their own symbol A. To avoid an ambiguous parse, it is preferable that the symbol A only capture non-empty argument lists and that empty lists are captured as a separate rule of the expression grammar.

Check your understanding: Attempt to make a context-free grammar using this information before moving on. Assume there is a terminal symbol named *id* which recognizes all variable

names and a variable symbol named *num* which recognizes all numeric literals. To write a keyword as a terminal symbol, put it in double quotes, e.g., “+” is a terminal symbol which matches only the string “+”. For a terminal matching the empty string only, write ““.

Solution: The context-free grammar for our example language is as follows. In our solution, A is a variable symbol for argument lists.

$S \rightarrow D S$

$S \rightarrow D$

$D \rightarrow id \ "=\> E$

$D \rightarrow id \ "("\> A \ ")" \ "=\> E$

$V \rightarrow num$

$E \rightarrow "let" \ D \ "in" \ E$

$E \rightarrow "if" \ E \ "{" \ E \ "}" \ "else" \ "{" \ E \ "}"$

$E \rightarrow E1$

$E1 \rightarrow E1 \ "+" \ E2$

E1 → E1 “-” E2

E1 → E2

E2 → E2 “*” E3

E2 → E3

E3 → id “(” A “)”

E3 → id “(”“)”

E3 → V

E3 → id

E3 → “(” E “)”

A → id

A → id “,” A

The rules for start symbol S indicate that a program is a series of definitions. Alternatively, one could make the first rule produce S S rather than D S, but D S is preferred because it gives an unambiguous parse, while S S is ambiguous between multiple parses which happen not to affect program meaning. The first rule

for definitions expresses variable definitions, the second function definitions. The rule for V expresses the only values: numeric literals.

The symbol E contains only the expressions with lowest precedence: “if” and “let”. Symbol E1 contains the low-precedence arithmetic operators “+” and “-”, and ensures they are left-associative. Symbol E2 contains the higher-precedence left-associative multiplication operator.. Symbol E3 contains the high-precedence expressions: atomic expressions like values and variables, as well as explicit parentheses. The rule for parentheses interacts with the precedence-climbing rules (such as $E \rightarrow E1$) to complete the treatment of precedence: once the grammar decides to parse higher-precedence symbol, it will only consider a lower-precedence symbol to be a child to the high-precedence symbol if there are explicit parentheses in between, which reset the precedence level. Argument lists A only capture nonempty argument lists and thus have only two rules. If we were to capture empty and nonempty lists in a single symbol A in a naive way, the grammar would allow optional commas at the end of every argument list, which we do not desire.

Software Tools

We give an overview of the state of software tool support for context-free grammars.

History of Tools

The prominence of context-free grammars in programming language design owes in part to their strong support in software development tools. Since at least 1975, *parser generator* software such as lex and yacc have respectively provided support for parsing terminal symbols using regular expressions and a large class of context-free grammars called lookahead left-to-right (LALR) grammars for parsing variable symbols. Performance has received significant attention in the development of these tools, and they use restricted classes of grammars because those classes have efficient parsing algorithms, where a naive breadth-first search of the rules can have unbounded runtime on inputs that do not parse. Even non-naive algorithms for general context-free grammars such as Earley parsers and CYK parsers can have cubic complexity. The parsers created using lex and yacc-like tools together tend to have optimal asymptotic complexity in contrast, parsing programs in linear time.

Over the years, implementations of parser generators have diversified significantly, with the popular open-source implementations *flex* and *bison* emerging in the 1980s, and with implementations for a wide array of programming languages emerging since then. For example, the prominent Java parser generator ANTLR emerged in the early 1990s. More recent parser generators have explored broader classes of context-free grammars such as Generalized LR (GLR) grammars, which provide polynomial-time, but not always linear-time performance. Over time, parser generators have also invested efforts in simplification of debugging. Despite the arrival of new algorithms and new tools with diverse feature sets, context-free grammars have remarkable longevity as the core foundation for all these tools, ranging from the 1970s up through the time of this book's writing.

Limitations of Tools

Though traditional parser generators based on classes of context-free grammars are foundational tools for the implementation of programming languages, every tool has limitations which should inform the choice among the available tools.

Expressive power

Expressive power has well-defined limits for each tool. Not every context-free grammar can be represented with the most performant algorithms, which require that it is possible to pick the next rule by looking at a constant number of symbols (often a single symbol). Even generic parsers for context-free grammars are limited by the fact that the grammars for some popular programming languages such as C and C++ are not truly context-free, and require additional implementation techniques in addition to the use of pure context-free grammars.

Though an understanding of this limitation is valuable for the Theorist, context-free grammars are flexible in practice. When programming languages include non-context-free aspects in their syntax, those aspects are typically limited and the context-free tools remain valuable in the implementation of such languages. Due to language designers' long collective history using context-free grammars, designers of new languages can typically ensure that the grammars of new languages are fully context-free, unless they specifically aim to provide familiar syntax for programmers who use existing non-context-free languages.

Ambiguity Debugging

A notable limitation of context-free grammars is that they fundamentally allow ambiguity, where a single string may be parsed in many different ways which potentially reflect dramatically different meanings of the program. Most programming language designers aim to avoid any ambiguity that would affect the meaning of a program, thus removing ambiguity is a common aspect of debugging the design of a context-free grammar.

While common ambiguities such as ambiguity of precedence and associativity have well-established fixes, debugging of these ambiguities can be challenging for new designers. This is particularly true for LALR-based parsers that present these ambiguities to the Implementer through the lens of parsing rule conflicts called *shift-reduce* and *reduce-reduce* conflicts. This terminology is specific to the LALR parsing algorithm and thus represents an additional educational hurdle atop the core notion of a context-free grammar.

Motivating Parsing Expression Grammars

The next chapter explores Parsing Expression Grammars (PEGs), a competing class of formal grammars. For the purposes of this textbook, the key motivating factor behind PEGs is that they are

incapable of possessing ambiguity. Compared to context-free grammars, PEGs are understood directly as parsing algorithms, with a direct understanding of how a grammar is executed, without parser generator algorithms as an intermediary. There is pedagogical appeal to such grammars because there is no intellectual barrier between the reader's understanding of their running implementation code and their formal grammar.

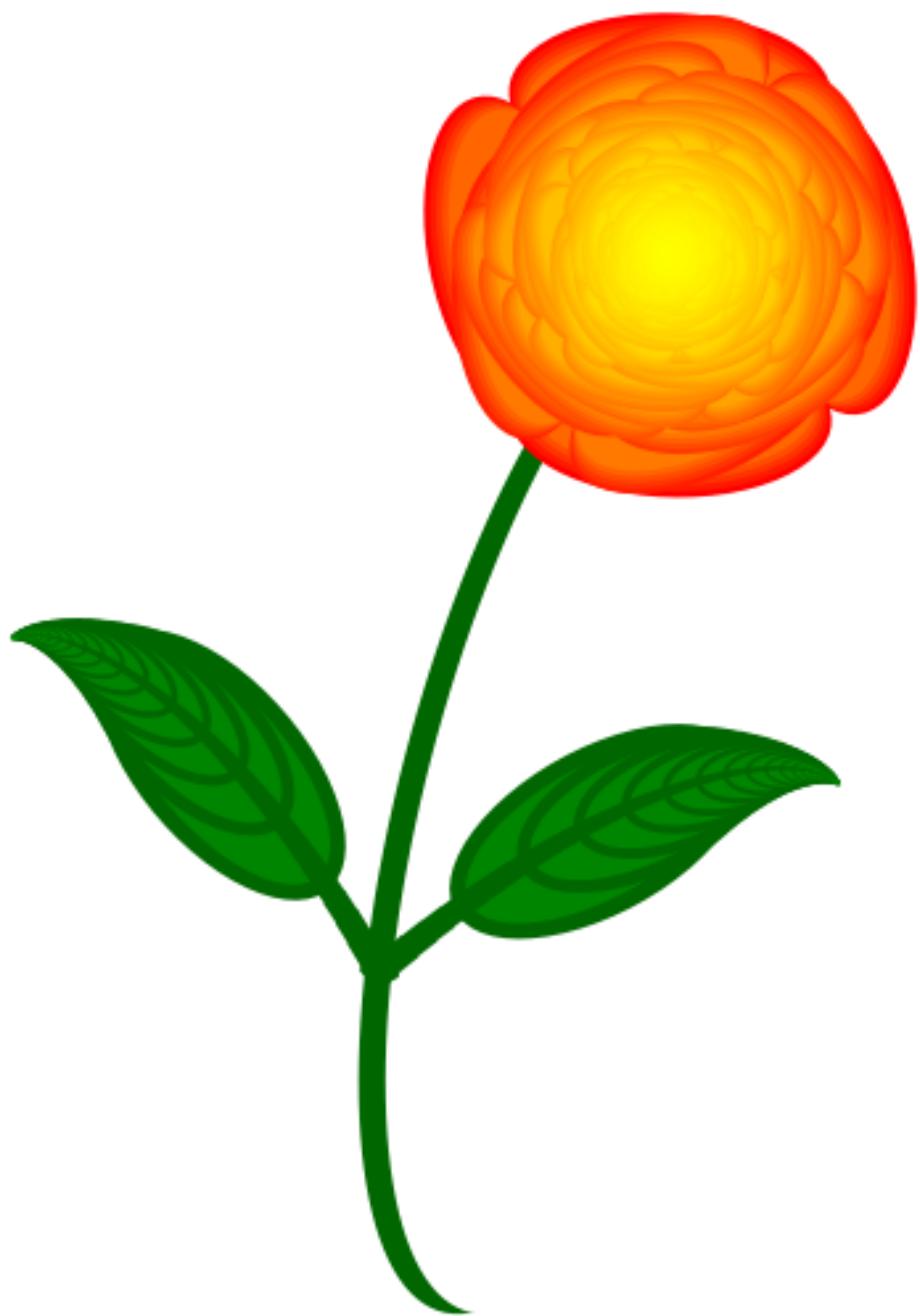
Context Free Art

The Context Free Art project is responsible for Context Free, a programming language and tool for procedural graphics generation using a generalization of context-free grammars to visual shapes. In place of symbols, Context Free considers shapes, using the following correspondence:

- * In place of symbols, there are *shapes*, which have names and have a variety of attributes attached to them such as x and y-coordinates, color, rotation, and scale. When a shape is created, these attributes can be adjusted, typically relative to the attributes of the parent shape from which it was produced.
- * In place of a start symbol, there is a named *start shape*. This tells Context Free which of the named shapes is meant to represent the entire image

- * In the place of production rules, there are *shape rules* which can recursively replace one shape with a collection of other shapes, apply specific drawing commands, or both
- * In place of terminal symbols, there are primitive shapes, such as triangle, squares, and circles.

We do not present Context Free in its entirety, for that purpose it is recommended to explore official documentation. Instead, we explore one of the provided examples, a rose, to explain the Context Free language by example. In the rose program, each shape has exactly one rule, though Context Free allows multiple rules. A rendering of the rose example is given below.



credit

Built-in Rose example, rendered in Context Free by the author

The rendering of the rose was produced using the following program:

```
startshape plant
```

```
shape rose1
```

```
{
```

```
loop 4 [r 90] petal [r -15...15 x 0.9 s 0.9 1.8]
```

```
rose1 [ r 14 s 0.92 hue 0.12 60 ]
```

```
}
```

¶

```
shape petal
```

```
{
```

```
CIRCLE []
```

```
CIRCLE [ s 0.975 x -0.025 hue 0.1 60 ]
```

```
CIRCLE [ s 0.95 x -0.05 hue 0.2 60 ]
```

```
CIRCLE [ s 0.925 x -0.075 hue 0.3 60 ]
```

```
CIRCLE [ s 0.90 x -0.10 hue 0.4 60 ]
```

```
}
```

```
shape stem
```

```
{
```

```
CIRCLE [ s 2 6 r -15 y -3 x -.5 ]
```

```
CIRCLE [ s 2 6 r -15 y -3 x -0.35 sat -1 b 1 ]
```

```
leaf [[ y -4 x -1.625 r 30 s 0.15 ]]
```

```
leaf [[ y -4 x -1.625 flip 80 r 30 s 0.15 ]]
```

{

```
shape plant
```

{

```
rose1 [ b 1 sat 1 ]
```

```
stem [ b 0.4 sat 1 hue 120 z -1 ]
```

}

```
shape leaf
```

{

```
loop 4 [[y 0.5 s 0.95 y 0.5 r 1]]
```

```
SQUARE []
```

```
finally {
```

```
CIRCLE [[r 7 s 7.5 15 y 0.5]]
```

```
CIRCLE [[r 7 s 7.5 15 b 0.2 y 0.5 s 0.80 0.90]]
```

```
leaf []
```

```
} }
```

The program defines five shapes:

- * Symbol **rose1** represents the flowering part of the rose
- * Symbol **petal** represents a single petal of the flower
- * Symbol **stem** represents the stem and its leaves
- * Symbol **plant** represents the entire drawing
- * Symbol **leaf** represents a single leaf attached to the stem

The first line indicates that **plant** is the start shape, i.e., the shape that should be rendered.

The definition of **rose1** indicates that creating the flowering part of the rose starts with creating four petals. Each petal is rotated 90 degrees from the last, plus an additional random rotation of -15 to 15 degrees. The **x** coordinate is adjusted as are the horizontal and vertical scales. Then a recursive call is made with adjustments in

rotation, scale, and hue. This recursive call could, in principle, repeat forever. The Context Free rendering process, however, will render a finite number of rule applications.

The definition of **petal** indicates that a petal is made up of five circles. The circle keyword is capitalized in the code to indicate that circles are a primitive shape, counterpart to the terminal symbols of a context-free grammar. The first circle has empty square brackets to indicate that no adjustments are made to the visual attributes; rather they are copied directly from the **petal** shape which is being converted to circles.

The definition of **stem** renders a stem as two circles. This counterintuitive approach is made clear by the use of minimum saturation and full brightness for the second circle, i.e., it is a white circle which obscures most of the previous green circle, leaving a sliver behind. Both circles are stretched so that the remaining sliver is elongated. Aside from these circles, the stem has two leaves attached, both rotated and scaled, one flipped.

The definition of **plant** defines the plant as the combination of the flowering part and the stem.

The definition of **leaf** first creates the connection between stem and leaf using squares, then uses two circles to create the vein and flesh of the leaf recursively. The **finally** keyword is used to ensure

that the flesh and veins start only after the first connecting piece is built.

In discussing Context Free Art, our goal is not to achieve proficiency. Rather, Context Free Art is a fruitful context for discussing the themes that arose in studying context-free grammars proper:

- * Expressive power is a key question throughout the design of a programming language, including the design of its grammar. Expressive range is the counterpart for assessing the expressive power of procedural generation. Context Free Art shows that the expressive power of context free grammars is significant: they can subdivide an image into parts (subdivide a symbol into a sequence of symbols), provide alternative rules, and express recursive and branching structures. Yet they are alike in their limitations too: the recursive structures they create are, in a colloquial sense of the word, self-similar, and they cannot create recursive structures where rules depend on the context surrounding a symbol.
- * Context Free Art, as with iterated function systems, is well-suited to creating fractals, where the same structures appear repeatedly, arbitrarily often, at different scales. This analogizes the structure of context-free grammars, where constructs such as expressions, values, and definitions will appear deeply nested within each other, but always following a consistent structure.

- * Ambiguity can be present in both Context Free Art and context-free grammars. In context-free grammars, ambiguity is often undesirable because it can make the meaning of a program ambiguous. If we translate the notion of ambiguous parsing directly to Context Free Art, it becomes less interesting: “the same image can be produced from the same start shape by following different rules”. Other notions of uncertainty do, however, hold up: the presence of randomness and alternative rules provides healthy variety in the generated images.
- * Both Context Free Art and context-free grammars have a similar relationship with infinity. Many context-free grammars describe infinite languages, i.e., languages with infinitely many different programs in them. Likewise, infinitely many bitmaps could be rendered from a Context Free Art program by applying different combinations of rules or stopping iteration at different points. In both cases, however, each program or bitmap is finite, built by applying one finite combination of rules from among infinite possibilities.

The reader is encouraged to take or leave each example and analogy as they see fit. Context-free grammars appear everywhere from programming languages to linguistics to procedural art generation. The beauty in this variety is that we need not master all three in order to master context-free grammars, and are free to explore most deeply the applications that speak to us.

Classroom Activities:

1. Draw parse tree diagrams when introducing examples, especially when explaining ambiguity
2. When exploring an ambiguous grammar, provide one parse of an ambiguous string and prompt students for a second parse.
3. Have every student write a declarative English sentence following the syntax explored in this chapter. Mix up the sentences and have each student draw the parse tree of a sentence.
4. If you choose to integrate Context Free Art in your course, give the students time in class to work in groups, editing code in Context Free and sharing the results with their neighbors. Unless additional instruction in Context Free is provided, it is recommended to provide example programs as a starting point and encourage modification of those programs
5. Give students a grammar and a series of strings. Ask them to predict which strings match the grammar or not.
6. Give the students two potential parses of a string. Give them an unambiguous grammar and ask them to predict which parse is produced by the grammar.
7. Give students an unambiguous grammar and strings that belong to it. Ask them to draw parse trees for the strings.

8. Give students a parse tree for a given string and grammar, then ask them to reconstruct a trace corresponding to the tree
9. Conversely, ask them to construct the parse tree corresponding to a given trace. There will be exactly one correct answer.

Exercises:

1. Implement a datatype for context-free grammars in your language of choice
2. Implement a datatype for derivations
3. Implement a program which consumes a context-free grammar, a derivation, and a string, returning true if and only if the derivation is valid derivation that the given string matches the given grammar
4. Implement a breadth-first search algorithm that searches for a derivation that a given string belongs to a given context-free grammar (it will not terminate if the string does not belong to the grammar)
5. Implement an algorithm which consumes a regular expression and produces a context-free grammar with the same language
6. Suppose I add an exponentiation operator E^E with higher precedence than both $*$ and $+$. How would you write the new grammar with precedence?

7. S-expression notation is a prefix notation originating with the Lisp family of languages, where each operation is enclosed in parentheses, with the operator name right after the opening parenthesis and the operands separated by spaces.
 - a. Write an context-free grammar for the polynomial example language, but changed to use S-expression notation
 - b. Do the same for the example functional programming language
 - c. Write a program which parses S-expressions. S-expressions are simple enough to parse that parser-generator software is typically not required for this task.
 - d. Write a discussion of how prefix vs. infix notation affects the complexity of parser design and implementation.
8. Consider the following grammar for well-matched strings of parentheses: $S \rightarrow " "$, $S \rightarrow LP \ S \ RP$, $S \rightarrow S \ S$. Derive the string of terminals “LP LP RP RP LP RP”, which corresponds to the string of characters “(())()”
9. Write three polynomial expressions of your own. Use every piece of the syntax at least once. Write derivations for each of your polynomials.
10. Given a fundamentally ambiguous language and grammar, write two derivations that prove its ambiguity. An example of a fundamentally ambiguous language is the set of strings $a^n b^m c^p d^q$ such that either (a) both $n=m$ and $p=q$ or (b)

both $n=q$ and $m=p$. Perform the exercise on this language and the following grammar, which recognizes it:

- a. $S \rightarrow S1$
 - b. $S \rightarrow S2$
 - c. $S1 \rightarrow A B$
 - d. $A \rightarrow "a" A "b"$
 - e. $A \rightarrow ""$
 - f. $B \rightarrow "c" B "d"$
 - g. $B \rightarrow "$
 - h. $S2 \rightarrow "a" S2 "d"$
 - i. $S2 \rightarrow D$
 - j. $D \rightarrow "b" D "c"$
 - k. $D \rightarrow "$
11. Given a grammar which is ambiguous but not fundamentally ambiguous, provide two derivations of the same string which prove the grammar is ambiguous:
- a. Rules $S \rightarrow S + S$ and $S \rightarrow "a"$
 - b. Rules $S \rightarrow S1$, $S1 \rightarrow S$, and $S \rightarrow "bc"$
 - c. Rules $S \rightarrow S < S > S$, $S \rightarrow S < S$, $S \rightarrow S > S$, and $S \rightarrow "d"$

12. Rewrite each grammar from the previous problem to be unambiguous but have the same string language. You do not need to prove that the answer is correct, as long as it is correct.
13. With your friends, come up with a grammar that generates funny sentences. Write some example sentences from the grammar
14. Documentation hunt: Context-free grammars are commonly used in official documentation of programming languages. Look up the full syntax of a programming language that you have used before. Practice reading context-free grammars by reading that syntax. What new features did you learn about in the process?
15. Prove that parse trees and parenthesized strings are equivalent by defining translations between the two and proving that the translations are inverses.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter6

Chapter 6

Parsing Expression Grammars

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * Parsing Expression Grammars (PEGs) are a style of formal grammars which, in contrast to context-free grammars (CFGs) are guaranteed to be unambiguous and have a clear operational meaning, i.e., are executed in a clear way
- * Precedence-climbing parsers are an approach to parser design and implementation which can be implemented using either CFGs or PEGs and can support a wide variety of programming languages
- * The crate “peg” implements PEGs in Rust
- * The library “fastparse” implements PEGs in Scala.

Context-free grammars are the foundation of parsing, yet their acceptance of ambiguous grammars introduces significant potential for ambiguity bugs because ambiguity is rarely a desired property in the syntax of a programming language. Parsing Expression Grammars (PEGs) answer the question: how can formal grammars be made fully unambiguous? The fundamental answer behind PEGs is that ambiguity arises in context-free grammars when there is an unconstrained choice between multiple rule applications in a derivation. Thus, PEGs eliminate ambiguity by discarding the unorder choice of rule applications in context-free grammars and replacing it with a prioritized choice: when a choice is given between two rules, whichever rule is written first must be tried first, and only once that choice is exhausted may the second choice be attempted. Prioritized choice means that PEGs can be understood directly as algorithms: a choice between two rules means “try the first rule, then try the second rule”. This clarity is helpful for debugging and for eliminating the possibility of ambiguity, but does not come for free. Not only do there exist context-free grammars which cannot be expressed as PEGs, but the straightforward execution semantics of PEGs makes it easy to write parsers which loop forever when one is not careful.

In the following sections, we present PEGs formally, discuss the precedence-climbing approach for parsers in the context of PEGs,

and summarize libraries for PEGs in several programming languages.

Defining PEGs

We first define the PEG syntax semi-formally, then give an example.

PEG Syntax and Informal Meaning

Just as a context-free grammar is a series of rules, a PEG is a series of syntax definitions. To distinguish PEGs from context-free grammars, we use left-facing arrows \leftarrow to define PEGs. We explain all the operators used in PEGs. We explain them in terms of how they execute. If matching a PEG against (the beginning of) an input string succeeds, then it may *consume* characters from the beginning of the string, so that subsequent PEGs will continue matching from the remainder of the string. Matching a PEG against an input string can also fail, in which any other choices are attempted until they are exhausted and the match fails as a whole.

- * At the highest precedence are the atomic PEGs. Any string in double quotes is a PEG which matches only that string. For example, the PEG “grammar” matches only the literal string “grammar”. Character classes, i.e., character sets and ranges,

are supported using square bracket notation as in regular expressions. For example, the PEG [a-z] matches any lowercase letter and the PEG [13579] matches the odd digits. The wildcard PEG, written as a period, matches any single character. Parentheses are used for grouping

- * The unary suffix operators have the second-highest precedence. The PEG $e?$ means that the PEG e is optional, i.e., it attempts to match e and, if that fails, the parse succeeds but consumes no characters from the input string. The PEG e^* means that e is parsed repeatedly, zero or more times. Specifically, e is matched repeatedly until it fails, then e^* succeeds and consumes the input characters which matched the successful instances of e . The PEG e^+ is like e^* except that the expression e is not optional and must be matched at least once.
- * The unary prefix operators have the third-highest precedence. Both unary prefix operators perform *lookahead*: they detect whether the beginning of the input string matches a given PEG, yet consume no input in doing so. The PEG $\&e$ succeeds if e matches the input string, and the PEG $\!e$ succeeds if e *does not* match the input string.
- * Sequential composition has the fourth-highest precedence. If e_1 and e_2 are PEGs, then the PEG $e_1\ e_2$ is their sequential composition. It executes by matching e_1 against the input string, allowing e_1 to consume input, and then matching e_2 on what remains. The sequential composition succeeds only if

both e1 and e2 succeed, consuming all characters consumed by e1 and e2.

- * Prioritized choice has the lowest precedence. The PEG e1 / e2 first attempts to match e1 against the input string. If e1 succeeds, then e1 / e2 succeeds, consuming the same input as e1. If e1 fails, then e2 is matched. If e2 succeeds, then e1 / e2 succeeds matching the same input, else it fails.

PEG Example

In the article that introduced PEGs, it is observed that PEGs can excel at certain parsing tasks that are challenging for parsers based on regular expressions combined with context-free grammars. Their driving example comes from C++, where angle brackets are used both for describing bitwise arithmetic operators and templates (e.g., for container types that can contain any type of element). When nested angle brackets appear in a template, the traditional approach makes it necessary to insert a space, such as in the following C++ declaration of a two-dimensional matrix of floating-point numbers:

```
vector<vector<float> > MyMatrix;
```

A traditional parser based on regular expressions and context-free grammars will process all terminal symbols in a greedy fashion before considering the nonterminal fashion. In a greedy

approach, the space between the closing angle brackets is necessary because otherwise “>>” would be recognized as a bit-shift operator, which later produces a parse error when the bit-shift operator can not be understood as part of the type syntax. If the syntax for expressions and types are defined using PEGs, this challenge is avoided. We present the PEG definition from the original article:

TemplType \leftarrow PrimType (LANGLE TemplType RANGLE)?

ShiftExpr \leftarrow PrimExpr (ShiftOper PrimExpr)*

ShiftOper \leftarrow LSHIFT / RSHIFT

LANGLE \leftarrow “<” Spacing

RANGLE \leftarrow “>” Spacing

LSHIFT \leftarrow “<<” Spacing

RSHIFT \leftarrow “>>” Spacing

The first definition indicates that a type (potentially containing templates) is parsed by first parsing a primitive type (i.e., type without templates), then optionally another type contained within angle brackets. The second definition indicates that an expression

(potentially containing bit-shift operations) is parsed by first parsing a primitive expression (i.e., expression without bit-shift operations) then parsing any number of additional expressions separated from it by bit-shift operators. The third definition parses a shift operation by trying left-shift first, then right-shift. The remaining four definitions define the terminal symbols. The “Spacing” definition, provided elsewhere in the article, allows any combination of whitespace and comments to come after any terminal symbol. A fully formal definition of a PEG should include this spacing explicitly, though we may sometimes omit it in this chapter when the spacing is implied.

Precedence-Climbing

Precedence-climbing is a classic and well-established approach for writing parsers for infix operators with precedence and associativity. The precedence-climbing approach applies equally well to both context-free grammars and PEGs. The grammar presented in the Context-Free Grammars chapter already implicitly takes a precedence-climbing approach, but this chapter makes the notion of precedence-climbing explicit and explores its implementation in PEGs.

In order to develop a precedence-climbing parser, one must first decide the set of operators to be parsed, their precedence, and

their associativity. Below is a table of operator precedences and associativities for the example parser for polynomials that we will develop in this chapter. In the table, precedence 2 is the highest precedence and precedence 0 is the lowest precedence

Operator	Precedence	Associativity
()	2	N/A
number	2	N/A
variable	2	N/A
*	1	left
/	1	left
+	0	left
-	0	left

A table of operator precedences and associativities for an example language

where “2” means highest-precedence and “0” means lowest precedence.

The precedence-climbing approach is described in terms of the precedence levels, e.g., precedence level 2 consists of all operations with precedence 2. The precedence-climbing approach is as follows:

- * Every precedence level has a variable symbol (cf. a definition in PEGs) to implement it.
- * Every operator belonging to the highest precedence level should consume at least one character of input. This is typically an easy assumption to fulfill, because terminals and parentheses have the highest precedence.
- * At all other precedence levels, parsing consists of two parts: parsing the operands and parsing the operators.
 - * For infix operators, the first operand will appear before the operator, so an operand is parsed first. When parsing operands, the precedence level increases. For left-associative parsers, this increase in precedence directly matches the approach used in the Context-Free Grammars chapter to develop unambiguous grammars. For right-associative grammars, this approach differs from the approach presented in Context-Free Grammars, but remains a convenient implementation approach for reasons discussed soon. For right-associative operators, the use of repetition PEGs makes it easy to collect all operands and apply the correct associativity.

- * To parse the operator, terminal symbols are used. If multiple operators are at the same precedence level, the parser for that precedence level should check all of them. The parser should use repetition to support multiple repetitions of the operators that belong to its precedence level. For example, the parser for precedence level 1 should allow parsing expressions of form “e1 * e2 * e3 * e4 * e5”
- * The start symbol for parsing an expression is the symbol with the lowest precedence.
- * Parsing is guaranteed to terminate because only finitely many steps of execution can occur until the highest precedence level is reached, at which point at least one character will be consumed. Termination is why it is important for the left operand of an operator to increase in precedence, even when the operator is right-associative.
- * Parsing is linear-time in the length of the input string when the number of precedence levels is fixed. This can be proved using the same approach used to prove termination. If we treat the number of precedence levels as a variable, the performance is proportional to the number of precedence levels multiplied by the length of the input string.
- * Parsing is guaranteed to respect precedence because the first step in parsing an expression is to recursively parse as many operations as possible at a higher precedence, then consume low-precedence symbols only once parsing of a high-precedence symbol has complete.

- * Precedence-climbing parsers are easily expressed as PEGs, which gives the guarantee that they are not ambiguous. Even when implemented as a context-free grammar, precedence-climbing parsers are written in a systematic way, which reduces risk of ambiguity bugs.

The beauty of precedence-climbing is that it works for any number of precedence levels, supports different associativities for different operators, can easily be integrated with parsers for non-infix operators, and is asymptotically efficient, taking only linear time in the length of a string, for a fixed number of precedence levels. A limitation is that although precedence-climbing parsers are asymptotically optimal, a naive implementation could be slower than alternate implementations by a constant factor, due to heavy reliance on recursion across the precedence levels.

Implementations of PEGs

Many programming languages provide implementations of PEGs. In functional programming languages, these implementations may be called *parser combinators* or *packrat parsers*, after the packrat library. We provide primers on using two different implementations of PEGs, one in Rust (the `peg` crate) and one in Scala (`fastparse`).

Rust

We provide a primer for the peg crate for Rust, which provides a specialized syntax for PEGs using a Rust macro. After introducing the PEG library, we explore an example implementation of polynomials.

To use the peg crate, add the following line to the dependencies section of Cargo.toml, with the version number 0.8.1 updated to account for any changes in version number:

```
peg = "0.8.1"
```

Then, at the top of your Rust file, write

```
use peg::*;
```

to import the entirety of the peg module.

The peg Syntax

The Rust peg crate implements PEGs using a macro, so that it can provide a completely custom syntax for PEGs, separate from standard Rust syntax. To create a parser, call the macro `peg::parser!{}` and write the parser code between the curly braces. Inside the braces, you can create a grammar, specify the

type of the input strings, and give the parser a name. For example, we write `pub grammar parser() for str {}` to indicate that we parse strings of type `str`, the parser is named `parser()`, and the parser has public visibility, making it callable from other files. Inside the curly braces, we write a series of rule definitions, which are the heart of a grammar definition.

A rule definition optionally starts with the keyword `pub` to make it usable from other files, followed by the required keyword `rule`. Next are the name, arguments, and return type, which follow the same syntax as a function definition. For the example in this chapter, all rules will have an empty argument list. After the return type come an equals sign and the body of the function. Thus all rules in this chapter have the form:

```
pub? rule rule_name() -> return_type = { body }
```

where the notation `pub?` indicates an optional keyword. The body of a rule is written using the following operations and others described in the official documentation:

- * *Symbols* are written using function call syntax, e.g., `id()` for an identifier symbol named `id`.
- * String literal PEGs are written using double quotes, so that “`+`” matches the literal string “`+`”.

- * Character ranges are written in square brackets using Rust notation for ranges and characters. For example, the PEG `['0'..='9']` matches any single digit.
- * Optional PEGs are expressed with postfix notation: `e?` is a PEG that optionally parses `e`, and succeeds without consuming input if `e` fails.
- * Repetition is written postfix: `e*` means any number of repetitions of `e`, including zero.
- * Non-zero repetition is written postfix: `e+` means any number of repetitions of `e`, except zero.
- * There is special support for repetition with a separator: `e ** s` means any number of repetitions of `e`, including zero, separated with `s`. For example, `id() ** ","` means any number of `id`, separated by commas.
- * Sequential composition is written by writing two expressions one after the other, separated by a space. Thus the sequential composition of `e1` and `e2` is written `e1 e2`.
- * Prioritized choice is written with a slash: `e1 / e2` means that `e1` is tried first, and `e2` is tried in its place if `e1` fails.
- * The peg crate allows capture, i.e., selecting substrings of a match which can be processed and returned as outputs of the parser. Capture `$e` captures the string that matches `e`.
- * Rust code can be included to process results of a parsing operation. The notation `x1:e1 ... xn:en {? body}` first

matches all e_i as a sequential composition, then assigns the names x_i to the result of each e_i , then runs the body, which is raw Rust code that can access the variables x_i . The symbol $?$ in this notation indicates that the parser returns a `Result` type in order to provide error handling, where a failure of any e_i results in a failed parse for the whole. Removing the $?$ removes the requirement of a `Result` type, but we will always use a `Result` type.

- * Positive and negative lookaheads are written prefix, respectively as `&e` and `!e`. These PEGs succeed (respectively fail) when PEG e succeeds, but do not consume any input, even when e does.
- * Parentheses can be used for grouping: `(e)` and e mean the same thing.

Example Implementation: Polynomials

We now consider a full implementation of our running example language: arithmetic expressions. The example is designed to follow a similar structure to the Scala example in the Scala section, which leads to minor differences from our naming conventions for precedence-climbing parsers. Notably, this example mixes parsing and evaluation: the parser returns the result of evaluating the expression. For complex parsers, the approach of evaluating programs in the parser will not scale, but

it is suitable in a toy example. We present the example with line numbers.

1 use peg::*;

2 peg::parser! {

3 pub grammar parser() for str {

4 pub rule numeral() -> i32 =

5 n:\$("-"? ((['1'..='9'] ['0'..='9']*) / "0"))

6 { ? match n.parse::<i32>() {

7 Ok(x) => Ok(x)

8 Err(_) => Err("i32") }}

9 rule atom() -> i32 = numeral() / ("(" b:expr() ")") { ?
Ok(b) })

10 rule op2() -> i32 = (l:atom() "*" r:op2()) { ? Ok(l *
r) }) / atom()

```
11     rule op1() -> i32 = (l:op2() "+" r:op1() {? Ok(l +  
r) })
```

```
12     / (l:op2() "-" r:op1() {? Ok(l - r) }) / op2()
```

```
13     pub rule expr() -> i32 = op1() {}
```

Line 1 imports the peg library, Lines 2-3 begin the definition of a parser named “parser”. Line 4 introduces a rule for parsing numerals into 32-bit integers. Line 5 captures a string starting with an optional minus sign, followed by either the numeral 0 or a string of digits starting with a nonzero digit, i.e., it prevents leading zeros. Line 6 converts the captured string into an integer and Lines 7-8 perform a type conversion between the Result type returned by `parse::<i32>` and the one expected used in the parser, supplying a dummy error message “i32” if needed. Line 9 parses atoms by parsing numerals where possible and parsing parentheses otherwise. Line 10 parses expressions of precedence 2 or higher by first parsing multiplications, then parsing atoms. Lines 11-12 parse expressions of precedence 1 or higher by first parsing additions and subtractions, then parsing higher-precedence expressions. Line 13 defines the expression parser to be the parser `op1()` defined on line 12.

This completes the definition of a parser for arithmetic expressions in precedence-climbing style.

Once the parser is defined, every rule with `pub` visibility can be called as a regular function call, and will return a `Result` containing the type indicated in the return type of the rule. To parse a string `s` as an expression, one would

call `parser::expr(&s)`. To extract the numeric result, one would

pattern match on the result:

```
match parser::expr(&s) { Ok(num)  
=> /* Use numeric result */, Err(e) => /* Handle parse error  
*/ }
```

Scala: Fastparse

We provide a primer on the fastparse library for Scala, which provides an efficient implementation of PEGs. After introducing the library, we explore an example implementation of polynomials.

If you use SBT for your Scala project, you need to add this line to your `build.sbt`, with the version number “2.2.2” updated to account for any changes in version number:

```
"com.lihaoyi" %% "fastparse" % "2.2.2" // SBT
```

Then, at the top of your .scala file, add the line:

```
import fastparse._
```

to access the fastparse module. Fastparse provides several built-in options for whitespace handling, and you should implement one by importing the appropriate module. For clarity, we perform manual white-space handling, which we indicate by writing the line:

```
import fastparse.NoWhitespace._
```

The Fastparse Syntax

Fastparse implements PEGs as a type `P[t]`, meaning “a parser that returns a value of type t upon success”. To create values of this type, a library of *combinators* are provided, i.e., operations which build up complex parsers from simple parsers. A parser is typically written by writing `P()` and then using the following combinator syntax inside the parentheses:

- * To write a string literal PEG, simply write a string literal in double quotes, like “myString” to match the literal string “myString”. Though this value is a plain Scala string, the fastparse combinators are often smart enough to take a string as input and interpret it as a parser. If a type error is

encountered, wrap the string with `P()` to explicitly convert it to a parser.

- * ‘CharIn’ provides character ranges, e.g., `CharIn("0-9")` implements the character range [0-9].
- * The values `Begin` and `End` correspond to the extended regular expressions ^ and \$
- * Repetitions are provided as a method call `e.rep` which generalizes the PEGs `e*` and `e+`. The rep method has an optional argument for the minimum number of repetitions: the default value of 0 corresponds to `e*` and a value of 1 corresponds to `e+`. A second optional argument allows inserting a separator between each repetition. For example, `e.rep(sep = ",")` parses a comma-separated list of `e`'s. Separators are often used for parsing list-like syntaxes including argument lists.
- * Sequential composition uses the tilde symbol (~). If `r1` is “my” and `r2` is “String”, then `r1 ~ r2` matches “myString”.
- * Prioritized choices use the bar symbol (|). That is, `r1 | r2` runs by attempting to match `r1` first, then attempting to match `r2` if `r1` fails.
- * *Capture* refers to returning a part of the input string as the result of a parser. Capture is not discussed explicitly in the theory of PEGs, but is essential in an implementation. Fastparse captures inputs with a method named “!”. Captured data can then be transformed with a method “map”, inspired

by a functional programming concept of the same name. Specifically, if x has type $P[t_1]$ and f is a function of type $t_1 \rightarrow t_2$ then $x.map(f)$ has type $P[t_2]$. $x.map()$ is understood as parsing a t_1 using x first, then applying f to the result.

- * Fastparse also provides an optimized version of sequential composition called *cut*, written $\sim/$. Cut commits to the current alternative of any surrounding prioritized choice. For example, suppose that the parser $e_1 \sim/ e_2 / e_3$ has successfully parsed e_1 , then it will continue as e_2 , without the possibility of trying e_3 when e_2 fails. The strength of cut is that it can improve both efficiency and error message quality. Its weakness is that it deletes all alternatives, including nested alternatives arising from recursive grammars, and should only be used in cases where no alternative is needed.

Example Implementation: Polynomials

We now consider a full implementation of our running example language: arithmetic expressions. Fastparse makes use of several pieces of advanced Scala syntax which this section intentionally brushes over. For example, functions that produce parsers should have `[_: P]` in the argument specification, an advanced style of type abstraction used by fastparse. A function named P is also called at the outside of every parser definition - this function P has a complex set of type overloads which are essential to type-checking fastparse parsers. In practice, not every parser needs this outer call to P , but when in doubt, call P .

The example is presented as it appears in the fastparse documentation, with minor differences from our naming conventions for precedence-climbing parsers. As in the fastparse documentation, this example mixes parsing and evaluation: the parser returns the result of evaluating the expression. For complex parsers, the approach of evaluating programs in the parser will not scale, but it is suitable in a toy example. We present the example with line numbers.

```
1 import fastparse._, NoWhitespace._
```

```
2 def number[_: P]: P[Int] = P(CharIn("0-  
9").rep(1).!.map(_.toInt) )
```

```
3 def parens[_: P]: P[Int] = P("(" ~/ addSub ~ ")" )
```

```
4 def factor[_: P]: P[Int] = P( number | parens )
```

5

```
6 def divMul[_: P]: P[Int] = P( factor ~ (CharIn("*/") .! ~/  
factor).rep ).map(eval)
```

```
7 def addSub[_: P]: P[Int] =
```

```
P( divMul ~ (CharIn("+-") .! ~/ divMul).rep ).map(eval)
```

```
8 def expr[_: P]: P[Int] = P( addSub ~ End )
```

Line 1 imports the fastparse library with no special whitespace handling. Line 2 defines a value called `number`, which is a parser for numbers, of type `Int`. That parser matches a nonempty repetition of digits, captures it (!), and maps the “to Integer” method (`_.toInt`) over the parse result. Line 3 defines a parser “`parens`” for parenthesized expressions, which wrap `addSub` expressions in parentheses. A cut `~/` is used after the opening paren because, if we see a paren, we know that no other rule could apply, and the expression must be a parenthesized one. Line 4 defines “`factor`” as “either `number` or `parens`”. Line 6 defines `divMul`, for expressions containing only division and multiplication (and parenthesized expressions). A `divMul` expression must contain at least one factor, optionally followed by any number of factors that are separated by operators. Here, `eval` stands for a function that evaluates the arithmetic expression. We omit the definition. Line 7 defines `addSub`, for all expressions of “add/subtract” level precedence or higher. Such expressions consist of “`divMul`” expressions, optionally separated by “`+`” and “`-`”. The top level function of the parser is “`expr`” on Line 8, which parses `addSub` against a whole string.

Together, these definitions are a precedence-climbing parser with built-in evaluation.

The discussion of the example parser would be incomplete without the discussion of its test cases. The following test case syntax would also be used when writing your own test cases.

```
9  val Parsed.Success(2, _) = parse("1+1", expr(_))
```

```
10 val Parsed.Success(15, _) = parse("(1+1*2)+3*4",
expr(_))
```

```
11 val Parsed.Success(21, _) = parse("((1+1*2)+(3*4*5))/3",
expr(_))
```

```
12 val Parsed.Failure(expected, failIndex, extra) =
parse("1+1*", expr(_))
```

Each line starts by pattern matching on a constructor for parse results. `Parsed.Success` is returned by successful parser calls. The first argument is the parsed value and the second value contains additional information. We only need the value, so we ignore the second argument using a wildcard pattern match.

The patterns specify exact result values (2, 15, 21), so Scala will report a (pattern-match) error if the parser gave the wrong answer, thus alerting us to any test failures. The fourth line instead returns `Parsed.Failure`, which reports what symbols were expected to come next after the failure point, the index (within the string) of the failure location, and extra information.

The syntax for invoking a parser is concise but advanced, thus we treat it as a black box. Let `exp` be a Scala expression of type `Parse[t]` for some `t` and let `str` have type `String`. Then the parsing syntax is:

```
parse(str, exp(_))
```

and returns a Success or Failure. If it returns `Success(x, y)`, then `x` has type `t`.

The last test case in the example shows how to use the debug tracing features, which print verbose information for parse failures.

```
13 val longAggMsg = extra.trace().longAggregateMsg
```

```
14 assert(
```

```
15   failIndex == 4,
```

```
16    longAggMsg ==
```

```
17    """Expected expr:1:1 / addSub:1:1 / divMul:1:3 /  
factor:1:5 / (number | parens):1:5, found """
```

```
18 )
```

Line 13 uses `extra.trace().longAggregateMsg` to produce a string containing detailed error messages that are useful for debugging. The next 4 lines perform two assertions, the first of which completes the test from line 12 and the second of which completes the test from line 13.

The user-friendliness of error messages depends on the details of how the parser was defined. Substantial experimentation may be required to achieve satisfactory error messages. Nevertheless, automatic error messages are a valuable debugging tool for the Implementer, even when effort may be required to make them suitable for the Practitioner.

Related Work

Our introduction to PEGs is based on the original article that introduced PEGs, “A Recognition-Based Syntactic Foundation” by Bryan Ford. The primers on implementing PEGs in Rust and Scala

are respectively based on the official documentation of the `peg` crate and the `fastparse` library. The author of `fastparse`, Li Haoyi, has also written a book on Scala titled “Hands-on Scala Programming: Learn Scala in a Practical, Project-Based Way.”

Classroom Activities

- * Repeat an activity about ambiguity from the Context-Free Grammars chapter in order to motivate the desire to remove ambiguity in PEGs.
- * Live-code the example parser with students following along.
- * Live-code a test case for the example parser and dedicate time in class for students to code their own test cases.
- * Give the students a precedence-climbing parser and a short expression. Have them draw out an execution trace of the precedence-climbing algorithm. The execution trace should list the precedence level before and after each character.

Exercises

- * In the programming language of your choice, define a datatype which represents PEGs.

- * A context-free grammar is *PEGable* if it can be represented as a PEG. Give one example of a context-free grammar that is *PEGable* and one that is not.
- * This chapter includes at least one example of a PEG that cannot be directly represented with regular expressions and context-free grammars. Name one such example.
- * Discuss: the Context-Free Grammar chapter discussed Context Free, a procedural art generation program based on context-free grammars. If Context Free were based on PEGs instead, what kind of drawings would get harder or easier to create with it?
- * Using one of the PEG libraries discussed in this chapter, write a parser for arithmetic expressions that use prefix “S-expression” notation, where each operation is wrapped in parentheses and the operator appears immediately after the opening parenthesis, with the operands separated by spaces.
- * Project: Using one of the PEG libraries discussed in this chapter, implement a parser for a functional programming language containing numerals, variables, parentheses, function calls, multiplication, addition, subtraction, and let-definitions of functions and variables.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter7

Chapter 7

Abstract Syntax Trees and Interpreters

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * We define the notion of Abstract Syntax Trees (ASTs) and explore their close relation to parsing and parse trees
- * We define the ASTs of an example language, without relying on a specific implementation language
- * We define environments, a key data structure for interpreting programs
- * We define an algorithm for interpreting ASTs, e.g., evaluating them, with relying on a specific implementation language

- * We explore implementation details for interpreters in specific implementation languages
- * We explore implementation techniques such as arenas, hash-consing, and out-of-band debugging information

The Implementer's ultimate goal in relation to programming languages is to compute the results of programs, i.e., to run them. Though parsing is an important part of the Implementer's job, it is ultimately in service of the goal of running programs. An *interpreter*, in its simplest form, is a program that takes in a program and returns its result, its value. Written as a mathematical function:

interpreter : Program → Value

This chapter culminates in defining an interpreter, realizing the Implementer's main goal. This chapter does not aim to explore every major implementation strategy: notably, we focus on interpretation rather than *compilation*, the approach of implementing a programming language by translating programs into another language for execution. We focus on interpreters for their simplicity.

The first section of the chapter addresses how to define the type Program for the purposes of an interpreter: Abstract Syntax Trees (ASTs). The second section introduces environments, which are

used to track the definitions of variables during interpretation.

The third section presents an interpreter for a toy language. The fourth section addresses implementation details for specific implementation languages.

Abstract Syntax Trees

Abstract Syntax Trees (ASTs) and Parse Trees (PTs) are both trees that represent parsed programs. In an implementation, there is often no distinction between ASTs and PTs, they are the same data structure. Instead, the difference is conceptual: PTs emphasize “What program structure is encoded in the source code?” whereas ASTs emphasize “How should program structure be represented in order to promote easy implementation?”

Algebraic Data Types

In strongly-typed functional programming languages, ASTs are typically implemented using a class of types known as Algebraic Datatypes (ADTs). ADTs combine three features into one language construct: inductive definition, tuples, and choice. ADTs take on specific names and properties in different languages, e.g., they are called `enum` in Rust, but we use the following language-neutral notation to describe ADTs:

```
ADT type_name =
```

```
| variant1(arg_type1, ..., arg_typeJ)
```

```
| ....
```

```
| variantN(arg_type1, ..., arg_typeK)
```

which creates a new type `type_name` and creates N functions called `constructor` functions, one for each variant. Every constructor function returns type `type_name` and its arguments are specified by the argument types from each line. The constructors may have different arguments from one another. Values of `type_name` are created by calling the constructors and taken apart using pattern-matching.

ADTs are widely used not only for implementing ASTs, but for recursive data structures more broadly. As supplementary examples of ADTs, we give definitions of lists and binary trees. For simplicity, these definitions assume all values are integers (type `int`).

```
ADT int_list =
```

```
| Empty ()
```

```
| NonEmpty (int, int_list)
```

This definition captures the notion that every list of integers is either the empty list or a non-empty list containing an integer followed by the remainder of the list. The three core features of ADTs are all present in this minimal example: the definition is inductive because NonEmpty lists contain another list, the argument of the NonEmpty construct uses a tuple of two arguments, and there are two variants for the type, providing choice. Lists are created using function call syntax,

e.g., `NonEmpty(1, NonEmpty(2, Empty))` represents the list `[1, 2]`.

Our language-neutral notation for pattern-matching defines functions over ADTs case-by-case. For example, the `length` recursively computes the length of a list like so:

```
length(Empty) = 0
```

```
length(NonEmpty(x, L)) = 1 + length(L)
```

The most common structure of the function definition follows the structure of the ADT: every case of the ADT corresponds to one case of the function and every inductive occurrence of `int_type` in the ADT definition corresponds to a recursive call in the function. This pattern remains true for more complex ADTs such as trees:

```
ADT int_tree =
```

```
| Leaf()
```

```
| Node(int_tree, int, int_tree)
```

This definition indicates that an `int_tree` is either a `Leaf` (the empty tree) or a non-empty `Node` containing a left subtree, an integer value, and a right subtree. The value `Node(Node(Leaf(), 1, Leaf()), 2, Leaf())` indicates the tree of depth 2 with the number 2 at its root, a left subtree containing only the number 1, and an empty right subtree. As an example function, the `depth` function on trees can be defined following the structure of the ADT:

```
depth(Leaf()) = 0
```

```
depth(Node(l, x, r)) = 1 + max(depth(l), depth(r))
```

As this example shows, a variant can have multiple inductive arguments, corresponding to multiple recursive calls which are combined to determine the result of the function.

From Grammars to Abstract Syntax Trees

To develop ASTs for a specific programming language, we start with the formal grammar of that language, typically a context-free grammar. We use the grammar in its simplified form, without concern for operator precedence and associativity. We do retain the distinctions between separate classes of syntax such as expressions and definitions. We present an example grammar for a toy programming language. In this example, we used condense notation for the grammar: for each variable symbol, we define all its production rules together, with the replacements of each rule separated by vertical bars. Our example contains separate classes for values (V), expressions (E), and definitions (D):

$V \rightarrow \text{num}$

$E \rightarrow \text{id} \mid \text{num} \mid E^{**} E \mid E^+ E \mid E^- E \mid \text{"let"} D \text{"in"} E \mid \text{id} (" E ")$

$D \rightarrow \text{id} = E \mid \text{id} (" id ") = E$

In this definition, numbers (integers) are both values and expressions: every value is an expression. Expressions also include variables, arithmetic operators, let-definitions, and function calls. Definitions include variable definitions and function definitions.

To define the AST of the programming language as a set of ADTs, we create one ADT for each variable symbol in the grammar and create one variant for each rule. The arguments for each variant follow the structure of its rule's right-hand side: terminal symbols have scalar data associated to them such as strings or numbers, while variable symbols correspond to inductive occurrences of the ADTs. Keywords in the grammar are forgotten when translating to variants: the role of keywords in parsing is rather to determine which variant applies for a given input string. Based on these rules, we present the ADTs for the toy programming language:

```
ADT value =
```

```
| NumberValue(int)
```

```
ADT expr =
```

```
| Id(string)
```

```
| Number(int)
```

```
| Times(expr, expr)
```

```
| Plus(expr, expr)
```

```
| Minus(expr, expr)
```

```
| Let(defn, expr)
```

```
| Call(string, expr)
```

```
ADT defn =
```

```
| Var(string, expr)
```

```
| Fun(string, string, expr)
```

The ADT `value` has single constructor `NumberValue(int)`, thus there is little practical difference between the types `value` and `int`. We mostly define `value` for illustrative purposes: in more complex languages with a variety of values, the `value` ADT would have multiple variants. The ADTs `expr` for expressions and `defn` for definitions follow the grammar closely: every identifier is represented as a string and every occurrence of E and D is represented by `expr` and `defn`.

It is worth noting that an `expr` can contain a `defn` and vice-versa, i.e., the types `expr` and `defn` are *mutually recursive*. When writing a function that takes in an expression, it is typical to write

a second function that takes in a definition. These two functions, as with the two types, call each other recursively.

The ADTs `expr` and `defn` are significantly more complex than simpler ADTs such as lists and trees. This is as it should be: these types encode the entirety of a programming language, and an entire interpreter for that programming language consists of a recursive function over each type. Given the complexity of what these types represent, they represent it quite compactly.

Environments

Implementing an interpreter will require an extra data structure called an *environment*. To understand what environments do and why they are desirable, we reflect on the meaning of one of the most fundamental constructs in programming: variables.

What does a variable mean? This question has two answers depending on whether the variable is immutable or mutable:

- * An immutable variable, as with mathematical variables, ranges over values. An immutable variable gets its meaning from substitution: a variable is something that can be replaced with its value everywhere in a program.

- * A mutable variable represents storage. A mutable variable gets its value from the state of a program, which can be modified as a program executes.

Corresponding to these two varieties of variables, there are two primary approaches to implementing variables:

- * Variables can be implemented by substitution. When a function call is evaluated, the values of its arguments can be plugged in for the parameter variables throughout the body.
- * Variables can be implemented by maintaining their state in a data structure called the environment.

These approaches have tradeoffs. Substitution simplifies mathematical reasoning about the behavior of programs, but is expensive in both time and space, potentially increasing the size of an expression by a significant amount. Environments, conversely, are more conceptually complex but provide a more efficient implementation.

Though environments E correspond most closely to the concept of a mutable variable, they can be used just as easily for immutable variables. Thus, in this chapter, we use environments to implement immutable variables. In the simplest case, an environment is a partial mapping (mathematical function) from variables to values, which is defined for a finite set of variables:

$E : \text{Var} \rightarrow \text{Value}$

In our case, environments are complicated by the fact that they can define both variables and functions, where function definitions specific a parameter name and a body expression. Thus the true type of our environments is

$E : \text{Var} \rightarrow \text{Value} \cup (\text{String}, \text{Expr})$

We use arrow notation to represent which variables are associated to which value: $x \mapsto v$ means that variable x has value v . We write concrete environments as sets of mappings, e.g., $\{x \mapsto 1, y \mapsto 2\}$ maps x to 1 and y to 2. We write $E(x)$ for the value assigned to a variable x , if any. In a slight abuse of notation, we write $E(f(x)) = e$ to mean that looking up f in environment E finds a function definition with parameter x and body expression e . We write $E[x \mapsto v]$ for an environment that assigns value v to variable x and otherwise behaves like E . We write Env for the set of all environments.

Interpretation Algorithm

This section presents an interpreter, a function from expressions to their values. The heart of the interpreter will consist of two mutually-recursive functions $\text{interp_expr} : (\text{Env}, \text{Expr}) \rightarrow \text{Value}$

and $\text{interp_defn}: (\text{Env}, \text{Defn}) \rightarrow \text{Env}$. The former computes the value of an expression and the latter computes an updated environment that incorporates the given definition. The top-level interpreter function merely calls interp_expr with an empty environment:

$$\text{interpreter}(e) = \text{interp_expr}(\{\}, e)$$

We give the complete definition of the interpreter, then discuss it:

$$\text{interp_expr}(E, \text{Id}(x)) = E(x)$$

$$\text{interp_expr}(E, \text{Number}(n)) = n$$

$$\text{interp_expr}(E, \text{Times}(e1, e2)) = \text{interp_expr}(E, e1) * \text{interp_expr}(E, e2)$$

$$\text{interp_expr}(E, \text{Plus}(e1, e2)) = \text{interp_expr}(E, e1) + \text{interp_expr}(E, e2)$$

$$\text{interp_expr}(E, \text{Minus}(e1, e2)) = \text{interp_expr}(E, e1) - \text{interp_expr}(E, e2)$$

$$\text{interp_expr}(E, \text{Let}(d, e)) = \text{interp_expr}(\text{interp_defn}(E, d), e)$$

$$\text{interp_expr}(E, \text{Call}(f, e1)) = \text{interp_expr}(E[x \mapsto \text{interp_expr}(E, e1)], e2)$$

where $E(f(x))=e2$

$$\text{interp_defn}(E, \text{Var}(x, e)) = E[x \mapsto \text{interp_expr}(E, e)]$$

$$\text{interp_defn}(E, \text{Fun}(f, x, e)) = E[f(x) \mapsto e]$$

To interpret a variable $\text{Id}(x)$, we look up x in environment E . Values are interpreted to themselves. Operators $*, +$, and $-$ first interpret the operands, then apply a mathematical operation. Let-definitions are interpreted by first interpreting the definition in order to update the environment, then interpreting the body expression in the updated environment. Function calls are interpreted by looking up the function definition $f(x)=e_2$ in the environment, interpreting the argument expression e_1 in the original environment, updating the environment with the value of the parameter, and finally interpreting the body. Variable definitions are interpreted by adding their values to the environment, whereas function definitions are added to the environment immediately.

Though the interpreter has many cases, each case is defined by interpreting subprograms recursively and combining their results, which enables us to reduce complexity. When a language has this property, we call it *compositional*.

Implementation Details

We discuss relevant implementation details for the languages Rust and Scala.

Rust

Implementation details for Rust include the representation of ADTs, the representation of environments, and the handling of input and output.

ADTs

In Rust, ADTs are implemented using the `enum` keyword. Though ADTs are a distinct concept from enumerated types (which have a finite set of values, each with its own name), enumerated types are a special case of ADTs, which justifies this naming convention.

An `enum` type in Rust must confront one important implementation detail that our language-neutral definitions of ADTs did not: in Rust, each `enum` must have a bounded size in memory, thus inductive references to the `enum` type must be made through indirection, e.g., through a reference. Rust has a wide variety of reference types that are useful in different scenarios, but for our purposes, the Box type is suitable.

We give lists of integers as an example definition.

```
enum int_list {
```

```
    Empty(),
```

```
    NonEmpty(i32, Box<int_list>),
```

```
}
```

Recursive functions with `int_list` parameters are easily written to use the `Box` type because it is dereferenceable using the `*` operator, just as references are. For example, the following Rust code computes the length of a list:

```
fn length(list : &int_list) -> i32 {
```

```
    match *list {
```

```
        int_list::Empty() => 0,
```

```
        int_list::NonEmpty(_, rest) => 1 + length(rest),
```

```
} }
```

In contrast, the syntax for constructing a list is made more verbose by the need for boxes. For example the list `[1, 2]` is written `int_list::NonEmpty(1, Box::new(int_list::NonEmpty(2, Box::new(int_list::Empty())))))` where the constructor `Box::new` creates a new box. Several strategies can be used to reduce the verbosity of this code. When appropriate, the constructors can be imported or given abbreviated names, so that the name `int_list` need not be repeated. When that is inappropriate or insufficient, custom macros can be implemented to provide custom, concise syntax, as is done with the standard library macro `vec![]` for creating vectors.

Environments

The Rust standard library provides a mutable Map data structure, but not an immutable Map data structure. If one wishes to use only the standard library, then the persistent environments described in this chapter can be simulated (inefficiently) by cloning a mutable Map when needed or (efficiently) by keeping track of changes made to the environment and reversing them when needed.

Instead, however, we explore a simpler implementation that relies on a crate from outside the standard library, the `rpds` crate. This

library provides persistent (immutable) data structures, including an immutable map type called `rpds:•HashTrieMap [K,V];`

provides built-in implementations of both mutable and immutable maps. The immutable map class `scala.collections.immutable.Map` is suitable for implementing environments. The Map type has two type arguments: `Map[K,V]` represents maps from keys of type K to values of type V. An environment is a map whose keys are identifiers and whose values include both function and variable definitions.

Scala provides extensive support for operator overloading, which is used in the Map class. If E is an environment and x is a variable name, then Scala allows functional call syntax `E(x)` to be used to look up the value of x. If E is an environment, x is a variable, and v is a value, then the update notation `E[x ↦ v]` is implemented in Scala using operator overloading: the arrow operator is overloaded so that `x -> v` indicates a key-value pair and addition is overloaded so that `E.+(x -> v)` means `E[x ↨ v]`.

Input/Output

Once the interpreter algorithm has been implemented, the interpreter needs a way to read in programs from the outside world and report back the result of a computation, i.e., it needs

input and output facilities. Input and output can be implemented with either either file-based or command-line-based approaches.

For file-based input/output, add the following lines to the top of your Rust file:

```
use std::fs::File;
```

```
use std::io::prelude::*;


```

```
use std::path::Path;
```

Once you have the name of a file you wish to read, you can

use `Path::new(name)` to create a path to the
file, `File::open(&path)` to open the file at that path,
and `file.read_to_string()` to extract the contents of the file.

Usually, the file name is provided to the interpreter as a
command-line argument.

For command-line-based input/output, add the following import
line to the top of your Rust file:

```
use std::io;
```

to import the standard I/O library. Then you can use the `read_line` method on `io::stdin()` to read a line of input from standard input.

To print the results to the user, the predefined `println()` function can be called. Command-line input and output are often combined into a read-eval-print loop (REPL) where a programmer writes a program one definition or expression at a time and observes the result of each one interactively.

Scala

Implementation details for Scala include the representation of environments and the handling of input and output.

Environments

The Scala standard library provides built-in implementations of both mutable and immutable maps. The immutable map class `scala.collections.immutable.Map` is suitable for implementing environments. The `Map` type has two type arguments: `Map[K,V]` represents maps from keys of type `K` to values of type `V`. An environment is a map whose keys are identifiers and whose values include both function and variable definitions.

Scala provides extensive support for operator overloading, which is used in the Map class. If E is an environment and x is a variable name, then Scala allows functional call syntax E(x) to be used to look up the value of x. If E is an environment, x is a variable, and v is a value, then the update notation E[x ↦ v] is implemented in Scala using operator overloading: the arrow operator is overloaded so that x -> v indicates a key-value pair and addition is overloaded so that E.+(x -> v) means E[x ↨ v].

Input/Output

Once the interpreter algorithm has been implemented, the interpreter needs a way to read in programs from the outside world and report back the result of a computation, i.e., it needs input and output facilities. Input and output can be implemented with either either file-based or command-line-based approaches.

For file-based input/output, add the following import line to the top of your Scala file:

```
import scala.io.Source
```

Once you have the name of a file you wish to read, you can now write `Source.fromFile(name).mkString` to get the contents of the file. Usually, fileName is provided to the main function of the interpreter as a command-line argument.

For command-line-based input/output, add the following import line to the top of your Scala file:

```
import scala.io.StdIn.readLine
```

then call the function `readLine()` to get a line of text from the user. To print the results to the user, the predefined `println()` function can be called. Command-line input and output are often combined into a read-eval-print loop (REPL) where a programmer writes a program one definition or expression at a time and observes the result of each one interactively.

Advanced Implementation Techniques

The techniques discussed this far are sufficient for a correct and complete implementation of an interpreter. However, production-grade implementations often incorporate additional implementation techniques for reasons such as performance or usability. We explore several such techniques.

Hash-Consing

Hash-consing is a programming technique for ADTs which ensures that every value of that type appears at most once in

memory. This approach can potentially save significant space if there is significant duplication between the values of the ADT. Moreover, because each ADT value is stored at a unique memory location, pointer comparisons suffice to compare ADT values, reducing comparison from a linear-time to constant-time operation. Though many interpreters do not need fast comparisons, they can be useful for more advanced program manipulations and for caching techniques. Hash-consing is not free: it makes every node of an ADT cost more space and increases the cost of constructing an ADT node. Thus, this approach should only be used when the performance gains from uniqueness and fast equality outweigh the memory and time costs associated with node construction.

Hash-consing gets its name from the use of hash functions in the constructors of values. Every ADT node includes a hash value which is a hash not just of that ADT node but all its descendant nodes as well. These hash values are used to maintain a global hash table of all extant values of the ADT. Each constructor function consults the hash values of constructor arguments to determine a tentative hash value for the constructed node. That hash value is used to consult the global hash table and efficiently determine if the new node has been created before. If so, the old node is returned, so that nodes are never duplicated. If the node

has never been created before, the new node is created and stored in the table.

As an example ADT, recall the type of binary trees of integers:

```
ADT int_tree =
```

```
| Leaf()
```

```
| Node(int_tree, int, int_tree)
```

to generate a hash-consed ADT for int_tree, we define two types: an ADT node including its hash value and one excluding its hash value:

```
ADT int_tree' =
```

```
| Leaf()
```

```
| Node(int_tree, int, int_tree)
```

```
and type int_tree = (hash, int_tree')
```

Now the type int_tree represents a tree with its hash value, and int_tree' represents the tree without its hash. Let Trees be a hash

table containing all trees, then the constructors can be implemented using the following pseudocode:

```
hash_cons(h, v) = if Trees.entries(h).contains(v)
```

```
    then (h, Trees.get(h).get(v))
```

```
else {Trees.insert(h, v); (h, v)}
```

```
Leaf() = hash_cons(hash(Leaf()), Leaf())
```

```
Node(l, x, r) = hash_cons(hash(l.hash, r.hash, hash(x)),  
Node(l, x, r))
```

The `hash_cons` function takes in the hash of a value and the value, checks whether the value is one of those associated with the hash `h` in the table, looks up the previous entry if so, and inserts the node otherwise. It returns `(h,v)` which has type `int_tree`.

In the `Leaf()` and `Node()` constructors, a method `.hash` is pseudocode for looking up the hash of an `int_tree` and `hash()` is pseudocode for hashing values. Thus, these functions look up precomputed hashes for subtrees, then combine them together with the hashes of numeric arguments in order to efficiently determine the hash of a node.

So long as all values of type `int_tree` are constructed through the constructors above, constant-type comparison is easy to implement:

```
eq((h1,v1), (h2,v2)) = (h1 == h2)
```

Arenas

Arenas provide a functionality called *region-based memory management*. An arena is a dynamically-created pool of memory from which values can be efficiently allocated, and which can be deallocated all at once. Arenas are well-suited for large batches of short-lived allocations, where the desire for fast allocation and even faster deallocation outweigh any potential memory overusage.

In programming languages with manual memory allocation, arenas can improve allocation speed because each element of the arena can be allocated with a simple address increment instead of by searching through free memory. Deallocation is made faster because the entire arena is deallocated with a single call. Additional speed benefits can potentially come from spatial locality: if AST nodes are allocated into a dedicated arena, then there is significant chance that nodes which are used together are allocated next to each other, which allows effective use of CPU

cache memory. Some arena allocators go further to reduce space: if *all* AST nodes are allocated through the same arena, then pointers to subtrees need not be represented as full addresses, only indices into the arena, reducing space usage.

A wide array of implementations of arenas are available. If you wish to use arenas, consult relevant libraries for your language. The pseudocode below highlights common typical usage patterns:

```
ARENA_SIZE = 100000;
```

```
arena : Arena<AST> = new_arena<AST>(ARENA_SIZE);
```

```
AST& e1 = arena.alloc(); AST& e2 = arena.alloc();
```

```
*e1 = AstNode(args...); *e2 = AstNode(args...);
```

```
...
```

```
arena.destroy();
```

In typical usage, the arena is initialized to a substantial size, to reduce the need for resizing. Storage for values is allocated out of the arena instead of a general-purpose allocator, then initialized and used for some computation. When the computation is done, the entire arena is destroyed.

Out-of-Band Metadata

How do we provide useful error messages when an error occurs deep within the implementation of a programming language, potentially after ASTs have been optimized, partially executed, or otherwise radically transformed? The standard solution is to augment every AST node with any metadata that are helpful for reconstructing a user-friendly error message, such as the source code locations from which it was constructed, allowing error messages to report the source of an error directly back to the programmer.

This approach works, but its potential impact on space usage should not be ignored. Suppose that the metadata of interest include a line number, starting column number, length in characters, filename, and directory, then the ADT variant for numeric literals would potentially grow in complexity from:

```
Number(i32)
```

to

```
Number(i32, i32, i32, i32, String, String) // val, line, col,  
len, file, dir
```

where the arguments to the old Number took 32 bits, the new version takes $32 + 32 + 32 + 32 + 64 + 64 = 256$ bits on a 64-bit processor, a factor-of-8 increase. This increase is notable not only because of concerns of memory usage, but because of time spent copying metadata and the loss of spatial locality when using large structures.

A first solution is to put metadata behind a reference in their own tuple or struct:

```
Number(i32, &(i32, i32, i32, String, String))
```

This provides a partial improvement: the new size is 32 + 64 bits for the main structure, a factor-of-3 increase over the original vs. factor-of-8. Because the metadata are stored behind a pointer, they can be copied efficiently when new nodes are made by reusing the pointer instead of deep copying.

For a greater improvement, however, we should recognize that generating error messages happens far less frequently than creating or traversing AST nodes. On the human scale, error messages are common and important, but there may be only a few error messages in a single compiler run for a project with thousands of files and hundreds of thousands of lines of code. Thus it is worth making error-reporting significantly more

expensive in time and memory in return for making the representation of ASTs efficient in the case there is no error.

Implementations of programming languages vary in their exact treatment of metadata, but we propose one example solution. One solution is maintain a global flag that indicates whether metadata generation is enabled or not. If it is not enabled, then all AST nodes are constructed efficiently without metadata. Once a compile error is encountered, the offending file is recompiled with line metadata generation enabled. To avoid the need for an entire separate AST data structure, the metadata are not stored in the AST nodes, but rather stored *out-of-band* in a global map from AST nodes to their metadata. Even then, if ASTs are duplicated and transformed throughout the implementation, it may be challenging to ensure that metadata are threaded properly throughout those changes. A potential way to ensure proper threading is for (e.g. copy) constructor functions of ASTs to check whether metadata generation is enabled and copy it, when so.

This approach could seem slow in the sense that an entire file containing an error may need to be compiled twice, but it is typically a net gain if the result is a performance increase that applies across the many error-free files of a project.

Classroom Activities:

1. Draw parse trees for programs first, then ask how best to represent trees in code
2. Walk through the transformation from formal grammars to ADTs
3. Live-code several cases of an interpreter, with examples
4. Present the interpretation algorithm without variables initially, and lead students to discover the need for environments

Exercises:

1. Translate the ADTs defined in this chapter into type definitions written the language of your choice.
2. Suppose you are given a parse tree using the following ADT, which is simple, but too flexible in that it allows writing parse trees which do not correspond to real programs: ADT pt = | Node(string, list<PT>). Where [] is the empty list, variables are represented as Node(the_variable_name, []) and numbers are represented as Node(the_number_as_a_string, []). All other operators are represented by Node(operator_name, operands). Assume the operator names are identical to the constructor names provided in this chapter. Now write a function which converts the `pt` type into the AST type. You

should support every variant described in this chapter, and you should return an error value if the parse tree does not correspond to any AST.

3. Extend the ADTs and the interpreter pseudocode to support core Boolean operators: true, false, if-then-else.
4. Extend your support of Boolean operators with *syntactic sugar*: write new, separate ADTs that include all the existing AST features, but also include the Boolean operators “and”, “or”, and “not”. Write a desugaring function which translates the sugared ADTs into the original ADTs by translating “and”, “or”, and “not” into if-then-else expressions. The function should recurse over the structure of the AST.
5. Repeat the above exercise, but this time, add support for exponents with fixed integer powers.
6. Desugaring presents a design tradeoff: it simplifies implementation of later stages of interpretation because it reduces the number of constructs they need to support. However, it complicates the creation of usable error messages for late stages of implementation because late-stage ASTs may look substantially different from source programs. Read the Out-of-Band Metadata section and write a brief design proposal (approximately 1 paragraph) for a solution that enables out-of-band source location information to be applied to desugared programs.
7. Implement the design you proposed in the previous exercise.

8. Refactor the ADTs so that there is a single variant for all binary operators. This variant typically takes an enumeration type as its argument to distinguish which operation is being applied. In a comment, describe whether you prefer this approach vs. the use of multiple variants, and why or why not.
9. Project: Translate the interpreter pseudocode in this chapter into the language of your choice.
10. Implement a function `bound_rename(x : String, y : String, t : AST) : AST` which assumes that the root node of `t` defines a new variable `x`, then renames `x` to `y` throughout `t`. If the variable `x` is defined more than once, only references to the definition at the root should be renamed. References to `x` are called *bound occurrences* if they appear in places where `x` is defined
11. Implement a function `free_rename(x : String, y : String, t : AST) : AST` which renames all free occurrences of `x` within `t` to `y`. Free occurrences are all occurrences except bound ones, i.e., occurrences where `x` has not been defined.
12. Implement a function `substitute(x : String, y : AST, t : AST) : AST` which replaces all free occurrences of `x` within `t` to `y`
13. For each of the following strings in the toy language, draw their parse tree:
 - a. `1 + 3`
 - b. `let x = 10 in x * x`

- c. let y = 3 in let f(y) = y*3 in f(z)
 - d. let a = let a(b) = a(a(b)) in a(1) in a
14. For each of the above trees that contain variables, add arrows to each variable expression's node in the tree, pointing back to the place that variable is defined. For variables which are defined more than once, be careful to select the correct definition.
15. For each of the above trees, compute its value by hand, or determine that it does not have a value (e.g., loops forever)
16. Project for the Social Scientist: Learn what design concerns for ASTs matter to developers of open-source and free-software compilers. Research public documentation and public conversations about the design of ASTs and intermediate representations (IRs). Long-running compiler projects such as the GNU Compiler Collection (GCC) carry significant experience about the different potential designs of ASTs and their impacts.
17. Review literature on topics related to ASTs and write a short report. Potential topics are listed:
- a. Compare ASTs to other intermediate representation (IR) formats for programs such as control-flow graphs, which are used throughout many compilers. What operations are easiest to perform on each representation? When do you need control-flow graphs and when do ASTs alone suffice?

- b. How do ASTs support the development of custom code-editing software, such as structured code editors?
- c. How does one formalize the notion of an AST for a program where a part is missing? How does one formalize edits to an AST? Potential keywords to search include holes, contextual modal type theory, zippers, and derivatives of a type.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter14
Chapter 8

Operational Semantics

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

What does the code mean? It is hard to think of a more fundamental programming question than this. For the Practitioner, Implementer, and Theorist alike, the meaning of programs is well-studied, and the formal word for the meaning of a program is its *semantics*. This chapter addresses operational semantics specifically. Operational semantics define what a program means by defining how it runs.

What phase of a programming language implementation do operational semantics correspond to? Consider the following phases of interpreters and compilers:

- * Parsing (processes the syntax of a program into an AST)
- * Type-checking (in statically-typed languages, analyzes the AST for typing errors)
- * Translation (in a compiler, converts a high-level AST into lower-level representations, eventually one directly that is executable by hardware)
- * Execution (running code, either in an interpreter or on hardware)

It is helpful to know that these different stages exist because behaviors (e.g. errors) encountered by a programmer correspond to different stages of an implementation:

- * Syntax errors arise from programs that are not syntactically well-formed, and are detected in the parser
- * Type errors, in a statically-typed language, are detected in the type-checker, before execution occurs
- * Translation errors, which most programmers only experience rarely, typically correspond to compiler bugs
- * Runtime errors such as exceptions occur during execution
- * Even after a program executes fully, a user may detect a mistake in its output

Operational semantics correspond to the final stage: execution.

Operational semantics for a high-level programming language correspond to interpreters and operational semantics for very low-level languages corresponds to running compiled code.

The Implementer could ask honestly: “Why are operational semantics necessary? Don’t they serve the same role as the interpreter I wrote?” In particular, the Implementer could be frustrated that the operational semantics in this chapter use a specialized, formal notation called inference rule notation, while an interpreter is written as code, a notation that is understood more widely.

Though the Implementer is correct that interpreters correspond to operational semantics, the Theorist has good reasons to prefer a formal inference rule notation:

- * The Theorist will write formal mathematical proofs about the semantics and its correctness. This work requires a concise notation that conveys a precise meaning in minimal space. In contrast, direct proofs about interpreters risk complicating the proof with irrelevant implementation details
- * The Theorist often studies the intimate relation between programming languages and formal logic. The inference rule notation originates in logic and enables the Theorist to explore this connection effectively

- * Inference rules are more general than code: while a traditional program has a single behavior for a given input, the Theorist often needs to describe situations where more than one rule could apply. Inference rules handle this situation well, but interpreters do not

For the other archetypes beside the Theorist, the main motivation for learning inference rule notation is multilingualism. A substantial body of literature about programming languages relies heavily on inference rule notation. By making

Vocab Reminder: An expression e is a program that we can try to evaluate. A value v is a program that is done evaluating.

The remainder of this chapter:

- * introduces inference rule notation
- * introduces big-step semantics, which evaluates a program all-at-once. Big-step evaluation is written $e \hookrightarrow v$, spoken “ e evaluates to v ”, and
- * introduces small-step semantics, which runs a program one step at a time. We write $e_1 \mapsto e_2$ for single small steps, spoken “ e_1 steps to e_2 ” (in one step). We write $e_1 \mapsto^* e_2$, for repeated small steps, meaning “ e_1 steps to e_2 ” in any number of steps

These operation semantics can be compared to the equation semantics $e_1 = e_2$, meaning that e_1 and e_2 produce the same value or same behavior.

Formal Notations

Inference rules are a notation for describing how a class of formally-defined mathematical assertions are proved. This notation is widely used for semantics and type systems in programming language theory, as well as in logic. Inference rules are closely linked to *judgements*.

Judgements

Definition: A *judgement* is a formal mathematical assertion that is defined using inference rules.

For example, we will define big-step operational semantics by defining a judgement $e \hookrightarrow v$ and define small-step operational semantics by defining a judgement $e_1 \mapsto e_2$. The word judgement can also refer to a specific instance of a general judgment, such as $1 + 2 \hookrightarrow 3$. We use the word judgement even when the assertion is not provable, such as $1 + 2 \hookrightarrow 4$, as long as $e \hookrightarrow v$ is defined by inference rules.

However, a mathematical assertion is only a judgement if it is defined using inference rules. For example, the assertion “Fermat’s Last Theorem is true” would be a true mathematical assertion, but not a judgement.

Many, but not all judgements include *hypotheses*, assumptions that are used during the proof of the judgement. The hypotheses are separated from the conclusion by a symbol called the turnstile (*hypotheses* \vdash *conclusion*), consisting of a vertical and horizontal line. The exact notation for the hypotheses varies by judgement, but is typically different from the notation used for conclusions. For this reason, it is not proper to think of the turnstile \vdash as fully equivalent to the concept of logical implication.

Inference Rules

Every rule is written with a horizontal line. There is exactly one judgement, called the *conclusion*, below the horizontal line. There are zero or more judgements, called the *premises*, above the horizontal line. When there is more than one premise, the premises are separated from one another by horizontal or vertical whitespace. If a rule has zero premises, we write a * above the line to make the absence of premises explicit. The name of the rule is written next to the rule, often at the left of the horizontal line. Some inference rules have requirements which do not

correspond directly to any formal judgements; these are called “side condition,” and can be written in parentheses to the right of the line. Thus the most general format for an inference rule is:

FirstPremise ... LastPremise

Name _____ (optional side condition)

Conclusion

In order to maximize compatibility with screen readers and to overcome typesetting limitations, the rest of this book uses a notation that is modified, but only slightly, so that it can be used interchangeably with traditional notation:

Rule Name

FirstPremise

...

LastPremise

Conclusion

(where *optional side condition*)

A rule asserts: if all the premises are provable (hold), then the conclusion is provable (holds). To prove the conclusion, apply this inference rule to the premises. The rule is called an inference rule because it is the rule *by which* we infer the conclusion from the premises. In the general format, we would read the meaning of the rule as

“by rule Name, we infer that when FirstPremise and ... and LastPremise all hold, then Conclusion holds“

An Operational Semantics Without Variables

Big-step operational semantics follow a similar structure to interpreters, so let's review the pseudocode for an interpreter and get inspiration for how to write the rules. This section considers a minimal interpreter *without* an environment, in order to provide a gentler introduction to inference rule notation.

```
interp : Expr -> Value
interp(n) = n
interp(e1 op e2) = interp(e1) _op_ interp(e2)
```

Minimal interpreter pseudocode without an environment

The first case says if e is already a value, then it evaluates to itself.

The second case says if e is an operation applied to operands we recursively evaluate the operands, then apply the operation to the operands to get the result. The underscores around `_op_` indicate it is a mathematical operation on numbers, not an expression.

Before writing down any rules, it's good that we skimmed the interpreter, because this tells us what the *key notions* are that we want to discuss while defining semantics. Next section, we make each of key notions into a *judgement*.

Defining Operational Judgements

In defining our judgements, we emphasize the following key concepts from the interpreter:

- * Is expression e is a value?
- * Does an expression e evaluate to a specific value v?
- * How are mathematical operations applied to values?

We define the first two concepts as judgements; the third is not a judgment but will appear in a rule.

- * Goal: **The judgement *e* value should hold if and only if *e* is a value**

Goal: The judgement $e \hookrightarrow v$ should hold if and only if e evaluates to v

Definition: Let “op” stand for any of the operator symbols, e.g., the symbols “+,-,*,/”. Then **op** (or _op_ in plaintext) stands for the corresponding mathematical operation on numeric values. If we wanted to be less precise, we could write “+” for both, but we prefer to keep these two ideas separate “a program that adds” vs. “the resulting sum”.

We now present the rules of the operational semantics. Assuming the language only contains numbers, there is a single rule for the judgment $e \text{ value}$. The rule NumIsVal proves that all numbers are values. Because it is sometimes self-evident which expressions are values from the syntax, many languages skip defining this judgement and simple use the letter v to indicate an arbitrary value. We define the judgement $e \text{ value}$ explicitly for the sake of completeness.

Rule NumIsVal

*

n value

The big-step judgment $e \hookrightarrow v$ consists of two rules. The first rule **EvalVal** means that a value evaluates to itself.

Rule EvalVal

v value

v

Putting together rules NumIsVal and EvalVal, one can conclude that every number evaluates to itself. To evaluate operations, we introduce a second rule EvalOp, which first evaluates the operands, then applies the operation to their values:

Rule EvalOp

$e1 \hookleftarrow v1$

 [Read](#) [Write](#) / [H...](#) / [1st](#) [rose.bohrer.cs@gmail.com](#) [About](#)  [Editing](#)
[Edition](#)

T X B I x_2 x^2 <> A A^a Aⁱ

A H H H ≡ ≡ ≡ ≡ — A „ „ < > 📸 📈 ⏱ ↪ ↫

e1 op e2 \hookrightarrow v1 op v2

(where op is one of +,-,*, or /)

This completes the definition of the judgements $e \text{ value}$ and $e \hookrightarrow v$ for our minimal big-step operational semantics. We finish this section with final remarks on inference rule notation.

Rule EvalOp is our first example of a rule with a side condition. Some authors may leave out the side condition in this definition, instead viewing op as a variable, which does not require a side condition. However, we take this opportunity to emphasize when a side condition is used: it is used when we need to state a requirement for applying the rule, but that requirement is not in the form of a judgement ($e \text{ value}$ or $e \hookrightarrow v$).

Just as a programmer should be careful to follow the syntax of their programming language, you should be careful to follow the syntax of each judgement carefully. For example, the variable names in $e \hookrightarrow v$ carry meaning: they mean that the left of the arrow should have an expression and the right should have a value. Anything else would be considered a syntax error, and not a valid premise or conclusion of a rule. For example, the symbols $2 + 3 \hookrightarrow 2 + 3$ are not valid syntax, because $2 + 3$ is not a value.

An Operational Semantics With Variables

We begin by reviewing the pseudocode of an interpreter that includes variables, implemented using an environment:

```
interp : (Env, Expr) -> Value
interp(E, v) = v
interp(E, x) = if E.contains(x) { E(x) } else { error }
interp(E, e1 op e2) = apply_op(op, interp(E, e1), interp(E, e2))
interp(E, let x = e1 in e2) =
    let v = interp(E, e1)
    let E' = E.add(x, v)
    interp(E', e2)
interp(E, let f(x) = e1 in e2) =
    let E' = E.add(f(x), e1)
    interp(E', e2)
interp(E, f(e1)) =
    let v1 = interp(E, e1)
    let (x, e2) = E(f)
    let E' = Env(x, v1)
    interp(E', e2)
```

Pseudocode for an interpreter that supports variables

At the heart of the interpreter is the type Env, which represents environments. Because this language treats functions and values as distinct, so does the environment. In order to define the

operational rules, we define a formal notation for environments, for use in judgements.

Environments

Recall from the lecture on interpreters that there are two primary ways to treat variables in an interpreter and in the operational semantics:

- * Whenever a variable definition occurs, immediately eliminate the variable by substituting in its definition throughout the program
- * Use an environment to keep a list of definitions and look them up upon use.

Each approach has different strengths. Substitution-style reasoning is often used when doing a proof about a program, but environment-style reasoning is more faithful to how most programming languages are implemented, and typically more efficient for use in an interpreter.

When defining a judgement, we do not define the notation for environments using a datatype definition, instead we define it using a context-free grammar:

$$E \leftarrow \cdot \mid E, x \mapsto v \mid E, f(x) \mapsto e$$

The variable E stands for “any environment”. A dot (\cdot) stands for an empty environment and commas separate the entries of an environment, with the dot omitted for non-empty environments. In the environment, an entry of form $x \mapsto v$ means that the variable x has value v , and an entry of form $f(x) \mapsto e$ means the function f is defined with body e , which can mention a parameter named x . Parentheses are usually used around examples of contexts for clarity, for example $(x \mapsto 3, f(y) \mapsto y^2 + 3)$ means variable x is defined as 3 and function f is defined by $f(y) = y^2 + 3$.

Defining the Operational Rules

Because environments are used in the evaluation judgment, the definition of the evaluation judgement must change. Evaluation is now written: $E \vdash e \hookrightarrow v$ which is pronounced “in environment E , expression e evaluates to value v “. The “turnstile” symbol (\vdash) simply separates the environment from the judgement $e \hookrightarrow v$. For the judgment to hold, E must define all the (free) variables used in e . For example, $(x \mapsto 3) \vdash x + 2 \hookrightarrow 5$ holds, but $(y \mapsto 3) \vdash x + 2 \hookrightarrow 5$, does not hold, nor does $(y \mapsto 3) \vdash x + 2 \hookrightarrow v$ for any value v ; the value is undefined.

We warm up by rewriting the variable-free operational rules to use an environment:

Rule NumIsVal

*

$$E \vdash n \text{ value}$$

When interpreting an inference rule, we interpret every variable as universally quantified (“for all”), which comes at the beginning of the rule. For example, rule NumIsVal means “for all environments E, for all numbers n, n is a value in environment E”.

Bear this interpretation of variables in mind when reading more complex rules.

Rule EvalVal

$$E \vdash v \text{ value}$$

$$E \vdash v \hookrightarrow v$$

Rule EvalVal means “for all environments E and all values v, if v is a value in environment E, then v evaluates to itself in environment E”

Rule EvalOp

$E \vdash e1 \hookrightarrow v1$

$E \vdash e2 \hookrightarrow v2$

$E \vdash e1 \text{ op } e2 \hookrightarrow v1 \text{ op } v2$

Check your understanding: Read aloud the meaning of the rule EvalOp, in the same style as the previous two rules

Answer: The rule is read as “For all environments E, expressions e1 and e2, and values v1 and v2: if e1 evaluates to v1 in environment E and e2 evaluates to v2 in environment E, then $e1 \text{ op } e2$ evaluates to $v1 \text{ op } v2$ in environment E.

As rules grow more complex, as in EvalOp, their meaning can get harder to parse. Once you understand this rule, which has multiple premises and an environment, you will be well-equipped to read the following rules. If the previous explanation is too clunky, an equivalent explanation is:

- * Let E be an environment and let e1, e2 be expressions.
- * Let v1 and v2 be the respective values of e1 and e2 in environment E
- * Then the value of $e1 \text{ op } e2$ is $v1 \text{ op } v2$ in environment E.

In the above rules, only the notation has changed, not the substance, because those variables do not concern variables. Having practiced the new notation on the above rules, we proceed with the novel rules for our extended language, starting with the variable rule:

Rule EvalVar

*

$E \vdash x \hookrightarrow v$

(where $E(x) = v$)

Rule EvalVar evaluates a variable, look up its value from the environment (written $E(x)$). Because variables are so fundamental, $E(x) = v$ is typically considered a judgement and is written as a premise rather than a side condition. We write it as a side condition in EvalVar only because we have not defined it.

Rule EvalVar evaluates a variable that has already been defined. In contrast, EvalLetVar defines a variable with a let definition, likewise for EvalLetFun and functions.

Rule EvalLetVar

$$E \vdash e1 \hookrightarrow v1$$
$$E, x \mapsto v1 \vdash e2 \hookrightarrow v2$$

$$E \vdash (\text{let } x = e1 \text{ in } e2) \hookrightarrow v2$$

Rule EvalLetFun

$$(E, f(x) \mapsto e1) \vdash e2 \hookrightarrow v$$

$$E \vdash (\text{let } f(x) = e1 \text{ in } e2) \hookrightarrow v2$$

EvalLetVar and EvalLetFun are the first rules that change E. In general, we change E whenever we define or redefine a variable or function. The EvalLetVar rule says that e1 should first be evaluated in the “current” environment E, then the resulting v1 is added to the environment as the definition of x while computing e2. The final result v2 of e2 is the result of the let. EvalLetFun similarly adds a function definition to the environment while executing the body, which only requires one premise.

To complete the operational semantics rules, we define a rule for function calls.

Rule EvalFun

$$E \vdash e1 \hookrightarrow v1$$
$$(x \mapsto v1) \vdash e2 \hookrightarrow v2$$

$$E \vdash f(e1) \hookrightarrow v2$$

(where $E(f(x)) = e2$)

Rule EvalFun evaluates the argument, looks up the function body, and evaluates the function body with the argument plugged in for the parameter.

Defining Errors

In the previous section, we only defined successful execution, not errors. The interpreter, in contrast, highlights a potential error case: evaluating an undefined variable. It is sometimes useful to define errors explicitly in the operational semantics by defining a new judgement $E \vdash e \text{ error}$ which means “program e produces a runtime error in one step, in context E”. The rule EvalVarErr defines it to be an error when an undefined variable is run.

Rule EvalVarErr

*

$E \vdash x \text{ error}$

(where x is not defined in E)

Check your understanding: Even after defining rule EvalVarErr, there are some expressions where undefined variable errors cannot be detected. Which expressions?

Answer: Any expression where the undefined variable appears in a subexpression, such as $f(x)$ or $z + 2$. This problem can be solved by adding additional rules which propagate the error from subexpressions to expressions. Defining such rules is an exercise.

Relating Judgements

To strengthen our knowledge of operational semantics, let's reflect on how the different judgements (\hookleftarrow , \mapsto , \mapsto^*) relate to each other. Note that \mapsto^* is defined as repeated stepping. The following example relates stepping to the equality judgement ($=$), which means that two programs have the same value/behavior:

Rule StepsEq

$e1 \mapsto^* e2$

$e1 = e2$

Rule StepsEq means if $e1$ steps to $e2$ in any number of steps, then $e1$ and $e2$ have equivalent values. Last time, we started to define the operational semantics of programs: how programs run.

However, we left out one of the most important features in all of programming: variables! Variables have a profound impact on the design of a language, including its semantics. Today, let's do a more complex operational semantics for a more complex language, with variables.

TODO

- * Make sure to include a trace example
- * Define small-step semantics too
- * Compare interpreter code across chapters for consistency

Related Work

Operational semantics, including small-step and big-step semantics, are only one of many competing styles of semantics. We list several other major styles of semantics:

- * Equational semantics ($e1 = e2$) define meaning by defining which programs are equivalent or not

- * Axiomatic semantics define meaning by defining what theorems can be proved about a program
- * Denotational semantics define the meaning of a program in terms of some other mathematical object, such as a Scott domain

Classroom Activities

1. Give students a form (either digitally or on paper) where they can fill in the definitions of key terms as they are introduced in class, to serve as a personal glossary
2. When reviewing values: write some expressions on the board, then circle the ones that are values
3. Write 3 expressions on the board and use step notation to write the traces on the board. Have students do the same for a different example program

Exercises

1. Write a rule that relates at least two of the judgements \hookrightarrow , \leftrightarrow , \mapsto^* , and $=$.
2. Define the extra error rules

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

*bookish.press/book/chapter15
Chapter 9*

Types

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

Introduction

Type systems, particularly static types, are one of the primary topics of interest for the Theorist. The Theorist's main approach for making programs more correct is to design an algorithm that will reject many buggy programs before they even run, i.e., "at compile-time". That algorithm is called a type-checking algorithm, and the mathematical rules that underly it are called typing rules. All of the typing rules, taken together, are called a type system.

The Theorist cares greatly about catching bugs *early*, before running programs. This priority can be explained by the idea that a single program might have many different potential inputs. Most bugs in production-quality software do not occur for all inputs, rather they occur when a user gives specific, often subtle or complex, inputs which the programmer did not anticipate. Thus, an essential question for correctness is “what type of inputs are acceptable?”. To illustrate the idea that our choice of input type is essential to correctness, consider the factorial function:

$$\text{fact}(0) = 1$$

$$\text{fact}(n) = n * \text{fact}(n - 1) \text{ otherwise}$$

Typically, an implementation of the factorial function is considered correct if it returns the factorial $n!$ for all inputs n .

Check your understanding: To determine if `fact()` is correct, we need one more piece of information. What information do we need?

We need to decide what *type* of number n is. If n is a natural number, then $\text{fact}(n)$ computes $n!$, so it is perfectly correct. If n is an arbitrary integer or arbitrary floating-point number, then $\text{fact}(n)$ is not correct: it loops forever whenever n is negative, and our definition of correctness forbids looping. Yet there is nothing

fundamentally wrong with this code, rather it is only correct when it is given the right inputs.

When the Practitioner chooses to use a statically-typed language, they are making a far more fundamental decision than whether to save a bit of time in catching bugs which they would catch anyway. They are deciding that specification matters to them as a fundamental part of programming. Catching qualitatively different bugs is often a key factor in this decision, but each person may have different reasons for using static types: in some cases they serve as valuable machine-checked documentation of code, in other cases they improve performance. No matter the reason, static types reflect a specific, fundamental way of thinking about programs: programs are worthy of being studied *statically*, because looking at a program's structure reveals insights separate from the insights we get when we run it.

Even if the Theorist cares deeply about static program reasoning, many Practitioners use static types for their everyday benefits. Types don't only help fix deep, mysterious bugs, but provide faster checking of everyday bugs. As an example, suppose we want to write a function that takes in a day of the week as a string and returns the next day of the week, but we forgot that weekends exist:

`next_day("Monday") = "Tuesday"`

`next_day("Tuesday") = "Wednesday"`

`next_day("Wednesday") = "Thursday"`

`next_day("Thursday") = "Friday"`

`next_day("Friday") = "Monday"`

Such a bug is almost certain to be caught during testing. It is hard to imagine testing the function without testing a weekend or at least a Friday. However, if we write the same function using a (static) enumeration type for the days of the week, and the type includes all seven days of the week, then the bug becomes easy to catch:

`next_day(Monday) = Tuesday`

`next_day(Tuesday) = Wednesday`

`next_day(Wednesday) = Thursday`

`next_day(Thursday) = Friday`

`next_day(Friday) = Monday`

The type checker can immediately determine that something is wrong, because the program does not define `next_day(Saturday)`

and `next_day(Sunday)`; the structure of the type reveals the structure of the bug. Though conceptually shallow, these bugs of omission can easily occur when copy-pasting code, when the definition of a type changes, or when performing more complex case analyses, such as casing on multiple ADTs at once.

Type checkers can also catch other common blunders which do not fundamentally depend on the structure of types, such as typos, e.g., in the following buggy factorial function:

`fact(0) = 1`

`fact(n) = n * fct (n - 1) otherwise`

where the recursive call to “`fact`” is misspelled as “`fct`”. Type checkers simultaneously check that variables and functions are defined before use, and in doing so they catch typos. Catching shallow errors is not an issue of shallow importance: the complexity of production software can make it time-consuming to trace down the source of even simple bugs, and some dyslexic programmers have even proposed spell-checking as an accessibility feature in casual discourse.

The Theorist not only cares about catching bugs early and catching them in every case, but catching them with certainty and mathematical rigor, at the level of a formal mathematical proof.

This way of thinking is justified by the fact that many software systems take on critical roles in society: if the code inside an airplane, medical device, or voting machine fails, it could have dramatic impacts on safety, health, and social stability, respectively. High-stakes programs deserve extensive, exhaustive efforts to ensure program correctness rigorously, yet these rigorously-developed type systems pay dividends even when implementing low-stakes software.

In summary, the Theorist's use of types reflects three values: the importance of program correctness, the importance of making correctness guarantees universal (i.e., making them apply to every program in a given programming language), and the importance of rigorous proof. These values are concisely summarized in Wadler's concept of "Theorems for Free," the idea that certain theorems about the behavior of a value fall directly out of its type. For example, it is a theorem that if a polymorphic function f has type $t \rightarrow t$ for all types t , then f is the identity function, and $f(x) = x$ for all x .

Type Safety

Most theorems about programming languages are not free, but they do serve to give correctness guarantees to every programmer for free. The most common kind of theorem is a Type Safety

theorem, which says that the execution of every well-typed program is well-behaved, as summarized in Robin Milner's phrase "Well-typed programs cannot go wrong."

The definitions of "well-behaved" or "wrong" can vary greatly between languages:

- * In a programming language like Rust that is intended for low-level systems programming, Type Safety might prove that memory errors such as memory leaks, use-after free, and double-frees never occur. This common version of Type Safety is also called Memory Safety.
- * In functional programming languages, Type Safety shows that the process of evaluating a program to its value does not modify the type of the program.
- * In a tool such as Coq which is used for checking the correctness of mathematical proofs, Type Safety can have much stronger implications: Type Safety can imply that programs satisfy complex specifications about their correctness. For example, one could prove that a sorting function successfully sorts its input.

These different versions of Type Safety can be summarized in the idea that "Types are Predictions," meaning that if you look at the type of a program, you can accurately predict something about its result.

The rest of this chapter explores a basic version of Type Safety for functional programs: “If a program has type t and the program returns a value, the value is of type t “.

Example Programming Language

This section presents an example language used to explore types. It combines and extends several of the example languages from previous chapters, and is designed based on the following requirements:

- * It should have functions
- * It should have more than one type
- * One of the types should be the unit type, which is commonly used for side effects.

These requirements were chosen in order to ensure that the language covers a variety of core language features. We present the grammar of the example programming language, which is divided into values (v), expressions (e), definitions (d), types (t).

```
v ::= n | true | false | ()
```

```
e ::= v | x | e op e | f(e) | e;e | let d in e
```

```
d ::= x : t = e | f(x : t) = e
```

```
t ::= bool | num | unit | t → t
```

Of these, the grammar for types is new: there are infinitely many different potential types. The following sections explore each syntactic category of the language in greater detail.

Values

As usual, numbers `n` are values, their type is `num`. Today we add the Boolean type, so `true` and `false` are also values. We also add a type called `unit`, which is well-known in functional programming languages but not imperative languages.

Booleans `bool` are the type that has two values; `unit` is the type that has one value. We use `unit` when we want to say “no interesting value”. If you are familiar with C, `unit` serves a similar role to `void`.

Though we have functions we do not provide values for functions. In most functional languages, functions are values, but we’re not there yet.

Expressions

Many of the expressions are standard: every value is also an expression, variables are expressions, and operators applied to operands are expressions. We have several other important expressions: function calls ($f(e)$), sequencing ($e;e$), and a let-definitions. Note that we treat statements as a special case of expressions, specifically the statement $e;e$ is an expression.

Definitions

Because we do not treat functions as values, we need separate ways to define variables and functions. The definition $x : t = e$ defines the variable x of type t to be the value of expression e of type t .

The definition $f(x : t1) : t2 = e$ defines the name f to stand for a function of type $t1 \rightarrow t2$ whose argument is x , of type $t1$, and whose body is e , of type $t2$

Types

Our basic (base) types are Booleans (`bool`), integers (`num`), and the unit type (`unit`), which has exactly one value, written `()`. The grammar of types suggest we can also build up bigger types from

smaller ones, recursively. Specifically, the function type ($t_1 \rightarrow t_2$) represents functions from type t_1 to type t_2 , and we could nest the \rightarrow if we want.

The following are examples of function types:

- * $\boxed{\text{bool} \rightarrow \text{num}}$
- * $\boxed{\text{unit} \rightarrow \text{num}}$
- * $\boxed{\text{num} \rightarrow \text{num}}$
- * $\boxed{\text{num} \rightarrow (\text{num} \rightarrow \text{bool})}$
- * $\boxed{((\text{num} \rightarrow \text{num}) \rightarrow (\text{num} \rightarrow \text{bool}))}$

The latter two examples are examples of *higher-order functions* because they nest arrows (\rightarrow). Higher-order functions are a key language feature for functional programming. However, the remainder of this chapter assumes functions are not higher-order, in order to simplify operational semantics.

Type Contexts

We now move towards type-checking and typing rules, i.e., checking whether a program is well-typed, both in code and in inference rules. Recall that the operational semantics required an environment E which kept track of the values of each variable. To

implement type-checking, we will need a corresponding concept for types.

A typing context is like an environment for types; it assigns types to variables. Just like E is the variable name for environments, the typical variable names for typing contexts are the uppercase Greek letters Γ , pronounced “Gamma,” and Δ , pronounced “Delta.” These symbols may be challenging for readers that do not use Greek variable names frequently. If you struggle to write or remember these symbols, you are welcome to use their Latin counterparts C and D : after all, C is short for “context”.

Typing contexts are defined by the grammar:

$$\Gamma ::= \cdot \mid \Gamma, x : t$$

Meaning \cdot is the empty context, and you can always add another variable x and its type t to a context. We leave out the \cdot when writing a nonempty context. For example, the context containing two variables x and y of types t_1 and t_2 would be written $(x : t_1, y : t_2)$. Conceptually speaking, a context is a map from names to types, and it thus supports function-like notation: $\Gamma(x)$ means “the type of x , according to Γ ”. Note that unlike environments, typing contexts have no need to distinguish between variables and functions: if a function f has type $t_1 \rightarrow t_2$, it suffices to write $\Gamma(f) = t_1 \rightarrow t_2$.

Typing Judgements

We now undertake the process of assigning types to programs. To do this, we need new notation in the form of a new judgement, called a typing judgement. The typing judgement $\Gamma \vdash e : t$ for expressions means “in context Γ , expression e has type t ”. Recall that the meaning of the symbol \vdash , pronounced ”entails,” is similar to but distinct from the idea of implication: the \vdash in $\Gamma \vdash e : t$ means “if we assume the variables defined in Γ , it implies that $e : t$ holds”.

If the judgement $\Gamma \vdash e : t$ does not hold for any t , then we say e is ill-typed (in context Γ), i.e., not well-typed.

We must also define a typing judgement for definitions; this typing judgement differs slightly from the judgement for expressions. A definition does not evaluate to a value, rather it introduces a name (or perhaps multiple names). Thus, the typing judgement does not simply assign a type to a definition, rather it produces an entire typing context Δ , because every name defined by the definition has an associated type, and definitions in many programming languages are capable of defining more than a single name. Thus, the typing judgement for definitions is written $\Gamma \vdash d : \Delta$ and pronounced “Gamma entails d has type Δ ”.

Typing Rules

It is standard practice to describe type-checking algorithms not in pseudocode, but using inference rule notation. The rules that describe a type-checking algorithm are called typing rules.

Inference rule notation is rarely used outside of programming language theory and logic, but is widely used in these fields, and is worth learning because it is compact and precise, even compared to pseudocode. This chapter assumes knowledge of inference rule notation. For more information on inference rule notation, consult the Operational Semantics chapter.

We present inference rules for each feature of the language.

Values

We present typing rules for all values in the language. Just as values are “base cases” for execution (they evaluate to themselves in zero steps), the typing rules for values are “base cases” because they have no premises: the following values are immediately well-typed, as a single step of the type-checking algorithm. Thus, they have no premises, indicated by an asterisk, and immediately conclude that each value has its associated type.

TyNum

*

$\Gamma \vdash n : \text{num}$

(where n is a literal integer)

TyTrue

*

$\Gamma \vdash \text{true} : \text{bool}$

TyFalse

*

$\Gamma \vdash \text{false} : \text{bool}$

¶

TyUnit

*

$\Gamma \vdash () : \text{unit}$

Expressions

We present the typing rules for expressions other than values.

Recall that the context Γ is only used to manage the types of variables, so many of these rules will pass Γ along unmodified from the premises to the conclusion, if the rules are not concerned with variables specifically.

TyVar

$\Gamma(x)=t$

$\Gamma \vdash x : t$

TySeq

$$\Gamma \vdash e1 : t1 \quad \Gamma \vdash e2 : t2$$

$$\Gamma \vdash (e1;e2) : t2$$

TyOp

$$\Gamma \vdash e1 : \text{num} \quad \Gamma \vdash e2 : \text{num}$$

$$\Gamma \vdash e1 \text{ op } e2 : \text{num}$$

TyApp

$$\Gamma(f) = t1 \rightarrow t2 \quad \Gamma \vdash e : t1$$

$$\Gamma \vdash f(e) : t2$$

TyDef

$$\Gamma_1 \vdash d : \Gamma_2 \quad \Gamma_1, \Gamma_2 \vdash e : t$$
$$\Gamma_1 \vdash \text{let } d \text{ in } e : t$$

In rule TyDef, the notation Γ_1, Γ_2 stands for appending all of Γ_2 at the end of Γ_1 .

Definitions

We present both rules for definitions, respectively for variable and function definitions. For this simple language, both rules produce a context containing exactly one variable.

DefVar

$$\Gamma \vdash e : t$$

$$\Gamma \vdash (x : t = e) : (x : t)$$

DefFun

$$\Gamma, (x : t_1) \vdash e : t_2$$

$$\Gamma \vdash (f(x : t_1) : t_2 = e) : (f : t_1 \rightarrow t_2)$$

Type-Checker Pseudocode

We present a type-checking algorithm in pseudocode, based on the typing rules. In this pseudocode, assume the type `Context` represents maps from variable names to types and assume the keyword `error` reports a type-checking error to the user. The main type-checking algorithm requires a context C , so the top-level `check : Expr -> Type` function immediately calls a helper function `tc : (Context, Expr) -> Type` with the empty context, written `empty`:

```
check(e) = tc(empty, e)
```

The main type-checking algorithm `tc` is defined by cases on the expression:

```
tc(C, n) = num
```

```
tc(C, true) = bool
```

```
tc(C, false) = bool
```

```
tc(C, ()) = unit
```

```
tc(C, x) = C(x)
```

```
tc(C, e1 op e2) =
```

```
let (t1, t2) = (tc(C, e1), tc(C, e2)) in
```

```
if (t1, t2) == (num, num) { num } else error
```

```
tc(C, e1;e2) = let _ = tc(C, e1) in tc(C, e2)
```

```
tc(C, f(arg)) =
```

```
let (t1 -> t2) = C(f) in
```

```
if t1 == tc(C, arg) { t2 } else error
```

```
tc(C, let x : t1 =e1 in e2) =
```

```
if t1 == tc(C, e1) { tc((C,x : t1), e2) }
```

```
else error
```

```
tc(C, let f(x : t) : t2 =e1 in e2) =
```

```
let C1 = C,x:t1 in
```

```
let C2 = C, f:t1->t2 in
```

```
if tc(C1, e1) == t1-> t2 { tc(C2, e2) } else error
```

The first 4 cases assign types to values: numbers, Booleans, and the unit tuple. Next, the variable case looks up the type from C. The operator case is for numeric operators, so it requires that both operands have number type, and results in numeric type regardless of which operator was used. The sequential composition case ($e_1;e_2$) requires that both e_1 and e_2 are well-typed, but discards the type of e_1 and uses e_2 's type. The function application case looks up the type of the function, checks the type of the argument, and checks that the argument type matches the input from the function type. The type of the result of the function call is the return type from the function type. The let-definition case is divided into a subcase each for variables and functions, but in each case it adds the defined variable or function to the context and then proceeds to check the body.

Big-Step Operational Semantics

Small-step operational semantics, consisting of the judgements $e \text{ value}$ and $e \mapsto e'$, analyze the execution of a program one step at a time. The same execution could also be analyzed in another way: all at once. Big-step operational semantics are how we capture the

execution of a program all at once: the judgement $e \hookrightarrow v$ means that e evaluates to value v. Both small-step and big-step semantics solve the same task of defining how a program runs, but carry different tradeoffs. In general, big-step semantics are more concise, but small-step semantics are more successful at describing programming languages where not all programs terminate.

The full version of the big-step judgement is written $E \vdash e \hookrightarrow v$, pronounced “in environment E, expression e evaluates to value v”. By including an environment E in the judgement, we can determine the values of variables that appear in the expression e. The remainder of this section presents the rules for the big-step operational semantics.

The rule OpVal indicates that every value evaluates to itself.

OpVal

*

$$E \vdash v \hookrightarrow v$$

(where v is a value)

The rule OpVar indicates that a variable x evaluates to the current value of x in the environment E , written $E(x)$.

OpVar

*

$E \vdash x \hookrightarrow v$

(where $E(x) = v$)

The rule OpOp defines all of the binary operators. Here, “op” stands in for the syntax of any binary operator, and the bolded, italicized *op* stands in for the mathematical meaning of the same operator. This rule evaluates an expression $e1 \text{ op } e2$ by first evaluating $e1$ and $e2$ to their respective values $v1$ and $v2$, then applying the operator to the values $v1$ and $v2$.

OpOp

$E \vdash e1 \hookrightarrow v1$

$E \vdash e2 \hookrightarrow v2$

$E \vdash e1 \text{ op } e2 \hookrightarrow v1 \text{ } \mathbf{op} \text{ } v2$

The rule OpDefVar evaluates a let definition that defines a variable x . First, we evaluate the right-hand side e_1 of the definition to its value v_1 . We then extend the environment to define x as v_1 , then evaluate the body e_2 in the extended environment. The value v_2 of the body serves as the final result.

OpDefVal

$$E \vdash e_1 \hookrightarrow v_1$$

$$E, (x \mapsto v_1) \vdash e_2 \hookrightarrow v_2$$

$$E \vdash \text{let } x = e_1 \text{ in } e_2 \hookrightarrow v_2$$

The rule OpDefFun evaluates a let definition that defines a function $f(x)$. First, we extend the environment to define f with parameter name x and function body e_1 , then we evaluate definition body e_2 to a value v , which serves as the final result.

OpDefFun

$$E, (f(x) \mapsto e_1) \vdash e_2 \hookrightarrow v$$

$$E \vdash \text{let def } f(x : t_1) : t_2 = e_1 \text{ in } e_2 \hookrightarrow v$$

Rule OpFun evaluates a function call $f(e_1)$. In this rule, x stands for the parameter name. We start by evaluating e_1 to its value v_1 , then we extend the environment to define x as v_1 , then we evaluate the function body e_2 to a value v_2 , which is the final result.

OpFun

$$E \vdash e_1 \hookrightarrow v_1$$

$$(E, x \mapsto v_1) \vdash e_2 \hookrightarrow v_2$$

$$E \vdash f(e_1) \hookrightarrow v_2$$

(where $E(f(x)) = e_2$)

Proofs

When the Theorist develops a new type system, writing down the syntax and typing rules of a language are only a small fraction of their overall job. The greater challenge is to prove that a language is type-safe. This section explores proofs about type safety. Writing proofs about programming languages is a specialized mathematical skill, and depending on your background, not all readers may need to learn how to perform these proofs. However,

even readers who do not wish to perform proofs are encouraged to read this section, to learn more about how the Theorist would undertake a proof of type safety.

For reasons of scope, this section does not provide a complete lesson on how to perform proofs. For readers with prior mathematical background, we recommend reading other texts on programming language theory to develop the skill of writing programming language proofs. For readers without prior mathematical background, we additionally recommend reading texts about mathematical proofs in general. That said, this section aims to include discussion that is accessible to readers without experience in proofs.

What Are Programming Language Proofs Like?

There are many styles of mathematical proof with different levels of detail and rigor. Programming language proofs are known for being particularly detailed. In large part, this is the case because type theory is intimately related to logic and the foundations of mathematics, so type theory proofs are likewise low-level, working directly with mathematical foundations! This style of proof is typically appreciated by people who find comfort in understanding every last detail of how a system works.

Theorists continue to develop new type systems everyday. Unlike working through the well-established proofs presented in this textbook, proving type safety for a real, new programming language design means proving safety for a design that is a work-in-progress. Thus the process of proving type safety is typically highly iterative: when a proof fails, the designer does not give up, but rather goes back, revises the design of the typing rules, and tries again until the proof succeeds. In programming language design, formal mathematical proofs are not a mere formality, instead they are a key tool for debugging the design; their payoff for the designer is telling them when and how to revise the typing rules.

High-level Proof Architecture

First we need to decide which kind of Type Safety theorem we want to prove. There are countless versions of the theorem, depending on whether you use big-step vs. small-step semantics, whether the language allows for exceptions and infinite loops, etc.

For small-step semantics, a standard theorem statement is:

Theorem Statement [Small-Step Type Safety]

If $\cdot \vdash e : t$ then either e value or $e \mapsto e'$ where $\cdot \vdash e' : t$

This small-step type safety theorem is usually divided into two lemmas, called Progress and Preservation, which are the heart of the proof:

Lemma Statement [Progress]

If $\cdot \vdash e : t$ then either e value or $e \mapsto e'$ for some e'

Lemma Statement [Preservation]

If $\cdot \vdash e : t$ and $e \mapsto e'$ then $\cdot \vdash e' : t$

The Progress lemma states that every well-typed program either has finished execution or can take a step of execution. The Progress lemma states that when a well-typed program does take a step of execution, the resulting program has the same type.

For languages like the one studied here, the Progress proof mostly amounts to a case analysis, ensuring that we did not forget to write an operational rule for any of the features in our language. Though this step is essential to correctness, it often involves less creativity than a proof of Preservation does. For that reason, we focus on Preservation here.

We focus on Preservation by developing a Type Safety theorem for a big-step semantics, which does not require separating the

theorem into Progress and Preservation lemmas.

In developing a Big-Step Type Safety theorem, we take the opportunity to generalize the theorem statement: our small-step type safety theorem statement required an empty context, but we will now use a more general theorem statement. Any context Γ is allowed, as long as the environment E is consistent with context Γ . We write the judgement $E : \Gamma$ to mean E is consistent with Γ . We define the judgement $E : \Gamma$ to hold if every variable x defined by Γ is also defined by E , and every value in E has the type required by Γ , i.e., $E(x) : \Gamma(x)$ for all x in the domain of Γ .

Theorem Statement [Big-Step Type Safety]

If $\Gamma \vdash e : t$ and $E : \Gamma$, then $E \vdash e \hookrightarrow v$ and $\Gamma \vdash v : t$

Mid-level Proof Structure

The structure of a PL proof, though intimidating at first, is systematic. Most PL proofs use a technique called rule induction, which is a (very general) generalization of a typical proof by mathematical induction.

Let's review how to do proof by induction on natural numbers, then build up to rule induction.

Induction on Natural Numbers

We use a proof by induction on natural numbers if we want to prove some property (“the induction predicate”) for all natural numbers. Since our goal is merely to remember how induction works, let’s prove a very boring property: squares are nonnegative. A fully detailed inductive proof has the following parts:

- * State the theorem and its induction predicate $P(n)$
- * Prove the base case $P(0)$
- * Assume the inductive hypothesis $P(k)$ is true, where k is any unknown natural number
- * Prove the inductive step $P(k+1)$ using the assumption $P(k)$
- * Conclude by induction that $P(n)$ holds for all natural numbers n .

As an example proof, we prove that the square of a natural number is nonnegative.

Theorem statement: “For all natural numbers n , $n^2 \geq 0$ ”

Predicate: $P(k) = "k^2 \geq 0"$

Base case: Prove $P(0)$ by equational reasoning.

P(0)

$$\leftrightarrow 0^2 \geq 0$$

$\leftrightarrow 0 \geq 0$ (which is true by reflexivity).

Inductive hypothesis: Let k be a natural number. Assume $P(k)$.

Inductive step, prove $P(k+1)$. We write a line-by-line proof:

P($k+1$)

$$\leftrightarrow (k+1)^2 \geq 0 \text{ [definition]}$$

$$\leftrightarrow k^2 + 2k + 1 \geq 0 \text{ [algebra]}$$

\leftrightarrow (1) $k^2 \geq 0$ and (2) $2k \geq 0$ and (3) $1 \geq 0$ [because nonnegativity is closed under addition]

Claim (1) holds by the IH.

Claim (2) holds iff $k \geq 0$, which holds because k is a natural number.

Claim (3) holds trivially by arithmetic.

Thus " $P(k) \rightarrow P(k+1)$ " holds

By induction, $P(n)$ holds for all natural numbers n .

Induction works by proving base cases, then proving complex cases by building them up from base cases step-by-step. An inductive proof is a proof for all natural numbers, precisely because every natural number can be built up by repeatedly adding 1, starting from zero. For example, to prove that $P(3)$ holds, we would first argue that $P(0)$ holds, then apply the inductive step 3 times to observe that $P(1)$, $P(2)$, and $P(3)$ all hold.

If we want, we could use induction on natural numbers to reason about programming languages. If we draw a program as an abstract syntax tree, that tree has several properties that happen to be natural numbers, such as the height of the tree and the number of nodes in the tree. However, treating a program as a number only works when the number gets smaller at each step of the proof. This is the case for some simple proofs about programming languages, but not for more complex proofs. In order to do proofs about programming languages in a more general way, we need a much more general notion of induction.

Rule Induction

Rule induction is a highly general version of induction, which requires two essential insights:

1. Induction works on all inductively-defined data, not just natural numbers
2. The core judgements of a programming language, such as $e : t$ and $e \hookrightarrow v$ or $e \mapsto e'$, are all inductively defined.

For reasons of scope, we do not explore point 1) in full depth, e.g., we do not demonstrate how to do proofs about data such as lists or trees, and we encourage consulting external sources for additional information. To explore point 2), perform the following exercise:

Check your understanding: Write out two proof trees, a proof of $(\text{let } x = 3 \text{ in } x * x) : \text{num}$ and a proof of $(\text{let } x = 3 \text{ in } x * x) \hookrightarrow 9$. Now draw these as actual tree data structures, where every node corresponds to a proof step, labeled with the name of the rule used at that step.

The idea of rule induction is abstract, but we can make it more concrete by considering a specific judgement and defining an induction template that works for the given judgement. As an example, we produce an induction template for the judgement (E

$\vdash e \hookrightarrow v$). In this induction proof, we will need a way to talk about proofs of this judgement as data, because we are not just inducting on the shape of a program, but the shape of its proof. These formal proofs are called derivations, so we use the variable name D to mean “a formal derivation of the judgement $E \vdash e \hookrightarrow v$ ”.

The first item of the proof template is an induction predicate. Here, the induction predicate takes the form $P(D)$, not $P(n)$ nor $P(e)$. We define the induction predicate for big-step type safety theorem as $P(D) = "v : t"$ where D is a derivation of $E \vdash e \hookrightarrow v$ and it is separately assumed that $\Gamma \vdash e : t$ for some Γ such that $E : \Gamma$.

The next item of the proof template is the base case. However, in contrast to natural numbers, complex mathematical structures like proof derivations can have multiple base cases. To determine the base cases, we look through all the rules for $E \vdash e \hookrightarrow v$. We make one base case for every rule that has zero premisses. In this case, we have two base cases, one for the rule OpVal and one for the rule OpVar. To prove the base cases:

1. Prove $P(\text{OpVal})$, i.e., prove $P(D)$ holds when the derivation D consists of the OpVal rule.
2. Prove $P(\text{OpVar})$.

Next we create the inductive steps, of which there can be more than one. Each inductive step might have multiple inductive

hypotheses, which can differ for each inductive step.

For each remaining rule R (i.e., every rule that has at least one premise), we create an inductive step. Call the i'th premise of rule R P_i and call the conclusion C. We add an inductive hypothesis $P(P_i)$ for each premise P_i , then prove $P(C)$.

In our case the inductive rules are the rules OpOp, OpDefVal, and OpDefFun.

Once these five cases are proven, the inductive proof is complete and $P(D)$ holds for all proofs D of $E \vdash e \hookrightarrow v$.

Proof of Type Safety

Before we prove type safety, we state one standard lemma which will be used in the proof of type safety.

Lemma [Inversion] Consider every typing rule, which all have form

$P_1 \dots P_N$

C

Whenever the conclusion C holds, then all the premises $P_1 \dots P_N$ hold.

Proof: By rule induction. Left as an exercise to the reader.

This inversion lemma holds for most type rules in most type systems, but needs to be proved as a lemma, because there exist proof systems where this does not hold, for example, it does not hold when there are two rules that prove the same conclusion.

We prove the main type safety theorem by rule induction.

Theorem [Big-Step Type Safety]

If $\Gamma \vdash e : t$ and $E : \Gamma$ and $E \vdash e \hookrightarrow v$, then $\Gamma \vdash v : t$.

Case for rule OpVar:

We want to show $\Gamma \vdash v : t$ where $v = E(x)$. This follows immediately from the assumption $E : \Gamma$.

Case for rule OpVal

We want to show $\Gamma \vdash v : t$. Because $e = v$ in this case, the assumption $\Gamma \vdash e : t$ constitutes a proof of $\Gamma \vdash v : t$ as desired.

Case for rule OpOp

Recall that the rule OpOp concludes $E \vdash e_1 \text{ op } e_2 \hookrightarrow v_1 \text{ op } v_2$ from premises $E \vdash e_1 \hookrightarrow v_1$ and $E \vdash e_2 \hookrightarrow v_2$. Call the derivations of each

premise D1 and D2 respectively.

We assume the inductive hypotheses on D1 and D2, meaning we assume (IH1) $\Gamma \vdash v1 : t1$ and (IH2) $\Gamma \vdash v2 : t2$ where $t1$ and $t2$ are the types of $e1$ and $e2$. By inversion on $\Gamma \vdash e1 \ op \ e2 : \text{num}$, we observe $\Gamma \vdash e1 : \text{num}$ and $\Gamma \vdash e2 : \text{num}$, thus we can simplify the inductive hypotheses to (IH1') $\Gamma \vdash v1 : \text{num}$ and (IH2') $\Gamma \vdash v2 : \text{num}$.

Proceed by cases on op : the type num is closed under every operator. Thus, by (IH1') and (IH2'), $\Gamma \vdash v1 \ op \ v2 : \text{num}$ holds in every case, as desired.

Case for rule OpDefVal

Recall the premises $E \vdash e1 \hookrightarrow v1$ and $E, (x \mapsto v1) \vdash e2 \hookrightarrow v2$. Let D1 and D2 be the derivation of each premise, and IH1 and IH2 be the inductive hypothesis for each derivation.

By inversion on $\Gamma \vdash (\text{let } x : t = e1 \text{ in } e2) : t2$, we obtain (1) $\Gamma \vdash e1 : t$ and (2) $\Gamma, x : t \vdash e2 : t2$. We then apply IH1 and IH2 to get (3) $\Gamma \vdash v1 : t1$ and (4) $\Gamma, x : t \vdash v2 : t2$.

Note that IH2 was applicable because $(E, x \mapsto v1) : (\Gamma, x : t1)$ by (3) and the assumption $E : \Gamma$. Fact (4) is what we wanted to show, completing the case.

Case for rule OpDefFun;

Recall that OpDefFun has the premise $E, f(x) \mapsto e1 \vdash e2 \hookleftarrow v$. Write D for the derivation of the premise and IH for its inductive hypothesis.

By inversion on $\Gamma \vdash (\text{let } f(x : t1) : t2 = e1 \text{ in } e2) : t$, obtain (1) $\Gamma, x : t1 \vdash e1 : t2$ and (2) $\Gamma, f : t1 \rightarrow t2 \vdash e2 : t$, then apply IH1 and IH2 to get (3) for all $E' : (C, x : t1)$ such that $E' \vdash e1 \hookleftarrow v1$, we have $\Gamma, x : t1 \vdash v1 : t2$, and (4) $\Gamma, f : t1 \rightarrow t2 \vdash v : t$.

Note in the application of IH1 that we use the generalized form of the IH, which allows all premises except the induction predicate to be generalized with a universal quantifier.

Note in the application of IH2 that IH2 is applicable because $(E, f(x) \mapsto e1) : (\Gamma, f : t1 \rightarrow t2)$ by (3) and by the assumption $E : \Gamma$.

Fact (4) is what we wanted to show, completing the case.

Case for rule OpFun:

Recall the two premises $E \vdash e1 \hookleftarrow v1$ and $E, x \mapsto v1 \vdash e2 \hookleftarrow v2$ where $E(f(x)) = e2$. Let D1 and D2 be the respective derivations and IH1 and IH2 be the respective inductive hypotheses. By

inversion on D1 and D2, obtain (1) $\Gamma \vdash e1 : t1$ and (2) $\Gamma(f) = t1 \rightarrow t2$.

Recall the assumption that $E : \Gamma$, then by (2), have (3) $\Gamma, x : t1 \vdash e2 : t2$.

By IH1 and (1) have (4) $\Gamma \vdash v1 : t1$. Then $E, x \mapsto v1 : \Gamma, x : t1$, which enables applying (IH2) and (3) to derive $\Gamma \vdash v2 : t2$ as desired, completing the case.

This completes the induction and thus completes the proof of type safety.

Classroom Activities

- * Draw a typing tree for an example program
- * Draw a derivation tree for the big-step semantics of an example program
- * After presenting the typing rules, lay out the cases of the type-checking algorithm and prompt students to try writing the implementations of cases
- * Live-code part of the type-checker in class or give students time in class to code it themselves

Exercises

1. In the language of your choice, define an ADT which describes all possible typing derivations
2. In the language of choice, define an ADT which describes all possible derivations of the big-step semantics
3. Implement the type-checker pseudocode in the programming language of your choice
4. Write formal typing derivations for each of the following expressions:
 - a. $1 + 2 * 3$
 - b. $\text{let } f(x : \text{num}) : \text{num} = x + 2 \text{ in } f(5)$
 - c. $\text{let } f(x : \text{num}) : \text{unit} = () \text{ in let } g(y : \text{bool}) : \text{bool} = \text{false} \text{ in } f(2);g(\text{true})$
5. For each of the above expressions, write a formal derivation of its big-step semantics to determine the value of the expression
6. For each of the following ill-typed expressions, write a partial typing derivation until you arrive at the part that does not type-check, then write an X above the premise that could not be proved:
 - a. $1 + z * 3$
 - b. $\text{let } f(x : \text{bool}) : \text{num} = x + 2 \text{ in } f(\text{true})$

c. let f(x : num) : unit = () in let g(y : bool) : num = false in
f(2);g(true)

7. Prove the inversion lemma

8. Rewrite the preservation proof to use induction on the depth of the big-step derivation, when written as a tre.

9. Suppose natural numbers (type `nat`) have been added to our programming language, constructed with a value `zero`, a function `+1 : nat -> nat` and an operation `natcase(e1, e2, e3(x))` which checks whether `e1` is zero, evaluates `e2` if so, and otherwise evaluates `e3`, which can refer to a variable `x` that stands for the value of `e1`, minus one.

a. Design new typing rules for `natcase`

b. Prove type safety for `natcase`

10. Write an induction template for:

a. Induction over values of type `unit`

b. Induction over values of type `bool`

c. Induction over lists of integers

d. Induction over typing derivations

TODOs

TODO: give more examples of typing derivations of concrete examples

TODO 2: Should add proper mutable state.

TODO 3: Should add conditionals

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter9

Chapter 10

Programmers as Users

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * Programming languages can be understood through the lens of human-computer interaction (HCI), where the programmer is the user of a programming language
- * Personas are fictional characters used a design tool, to help designers explore how different users might interact with a computer system
- * Methods from the social sciences can be used to answer a wide variety of questions from designers about how programmers interact with programming languages
 - * Quantitative methods produce numbers as data, which can be interpreted using statistical approaches

- * Qualitative methods produce data other than numbers, which can be interpreted using non-statistical approaches
- * You can ask and answer your own questions about your own language design using the methods discussed in this chapter

Language as Interface, Programmer as User

The academic field of human-computer interaction (HCI) studies the relationship between humans and computers in a wide variety of ways, often using interdisciplinary techniques. Despite the breadth of HCI as a topic, however, one of the most fundamental concepts in HCI is the concept of an *interface*. In simple terms, an interface is where the human and computer meet. In this chapter, we view programming languages as interfaces. The decision to view programming languages in this way has major implications for how we think about a programming language, because it invites us to ask questions about a language which someone might normally ask about an interface.

As a warm-up question, we ask: “How do the parts of an interface translate to the parts of a programming language?”

Check your understanding: Try to define the users, inputs, and outputs of a programming language before reading further.

- * An interface has users. In the case of programming languages, programmers are users.
- * An interface gives the user ways to provide input. In most programming languages, the programmer provides input by editing a textual representation of the program.
- * An interface gives the user ways to receive output. In programming languages, executing the program is one way to receive output. In statically-typed languages, additional output may be provided during the type-checking stage.
- * Many but not all interfaces are used to edit some set of underlying data. In the case of programming languages, the data are the programs themselves.

Once we have established basic terminology about programming languages viewed as interfaces, we can start to ask questions about the effectiveness of that interface:

- * How ergonomic is the interface? If the user wants to perform a given task, how many actions are required to complete that task, and how much effort is spent on those actions?
- * Do the inputs fully represent the data? That is, does the syntax of the programming language enable to write arbitrary programs?

- * Do the outputs fully represent the data? That is, can a programmer distinguish two different programs from one another based on the output alone?
- * What unique assets and needs do every different group of programmers have?
- * How do the inputs and outputs of the interface align with the assets and needs of programmers? Does the interface make use of programmers' strengths while meeting their needs?
- * How do the outputs promote or hinder programmer self-efficacy, i.e., their ability to plan and execute a solution to a problem? For example, do the error messages of a compiler direct the programmer to fix relevant bugs in their program?

For most Practitioners, lived experience is enough to show that usability concerns exist when a programmer uses a programming language. Most Practitioners probably have a sense that some code is easier to read, understand, and edit than other code, and that programming language designers potentially have a role to play in this. Through studying programming languages as interfaces (often but not solely using methods from Social Science), we can expand the Practitioner's lived experience as a user into actionable knowledge that we can use to guide the design of a programming language.

Example: Unreadable Code

As an example of a difficult interaction with a programming language, read the following C code. Meditate on it. If you do not understand it, you have achieved the goal of this exercise.

Discuss: What does this code do?

```
float Q_sqrt( float number )
```

```
{
```

```
long i;
```

```
float x2, y;
```

```
const float threehalves = 1.5F;
```

```
x2 = number * 0.5F;
```

```
y = number;
```

```
i = * ( long * ) &y; // evil floating point bit
```

```
level hacking
```

```
i = 0x5f3759df - ( i >> 1 ); // what the  
fuck?
```

```
y = * ( float * ) &i;
```

```
y = y * ( threehalfs - ( x2 * y * y ) ); // 1st iteration
```

```
//y = y * ( threehalfs - ( x2 * y * y ) ); // 2nd iteration,
```

this can be removed

```
return y;
```

```
}
```

Answer: This code computes an approximation of $1/\sqrt{\text{number}}$.

Discussion: To what extent can the design of the C programming language be blamed for the difficulty of reading the above code, and to what extent must blame lie with the programmer?

Potential Answer: C carries partial responsibility because it provides a particularly low-level interface to floating-point

numbers. In contrast to many other programming languages, C makes it particularly easy to view a floating-point number as a sequence of bytes, which invites the low-level bit manipulations seen in this function. Though C does not in the least require programmers to write this complicated style of code, it invites such code.

Usability for Programming Languages

Among programming language designers, it is a common goal to design a language that is *usable*. However, the notion of *usability* could have many different meanings, depending on the purpose of a given programming language, thus it is not enough to know that we wish for our language to be usable; a deeper analysis of usability is needed. We begin this deeper exploration of usability by exploring different ways that the notion of usability can be defined in HCI. Once we know the definitions that are available to us, we are better able to pick a definition that suits our goals and design around that definition.

International Standard

The notion of usability has been defined and used in at least one major international standard, ISO 9241-11 (Ergonomics of human-system interaction). One option, which we will take, is to use the

definition of usability from this standard. An advantage of this approach is that the definition is widespread and has been widely studied.

Definition (Usability):

“The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.” (ISO 9241-11, Ergonomics of human-system interaction)

This definition highlights three key components of usability. To even **state** the question “is my programming language usable?” we must define the following:

- * Who are the users of the programming language?
- * What are the programmers’ goals?
- * What is the context of use?

Once we state the question “is my programming language usable?”, answering the question requires answering three sub-questions:

- * How effective are programmers in achieving their goals? (i.e., to what extent do they eventually succeed in their goal?)

- * How efficient are programmers in achieving their goals? (i.e., assuming the programmer eventually succeeds, how many resources such as time, effort, and money are expended in order to succeed?)
- * How satisfied are the programmers? (i.e., subjectively, how happy are they with their experience as a programmer in a given language?)

None of these questions are simple, and none should be studied without a critical eye. For example, different people might have very different notions of what it means do something well, or different notions of what constitutes a lot of work. Moreover, liking or disliking software depends on any number of factors influencing one's opinion.

The six points listed above are the foundation on which we study programming languages as interfaces. Next, we use personas to explore the question of who the users are. After a brief discussion of goals and context of use, we provide an overview of research methods that can be used to answer questions of effectiveness, efficiency, and satisfaction.

Personas

A **persona** is a fictional person used to guide the design of software. In group discussions among designers, personas helps

keep hypothetical discussions of user behavior more concrete by providing the designers with a clear image of their intended users. When personas are working as intended, they help designers roleplay as users and, by getting into character, base their designs on empathy for their users. Even the process of writing this fiction helps designers pause and consider relevant aspects of their user audience, such as:

- * How educated are the programmers?
- * What informal knowledge do the programmers have?
- * What skills and operational knowledge do the programmers have? (i.e., what can they do?)
- * How much programming experience do the programmers have?
- * What is the nature of that experience?
- * In what context are the programmers using the programming language?

By looking at all your personas side-by-side, you can additionally explore:

- * How broad or narrow is your audience? Does it include everyone that you wish to include?

Though personas are effective tools for exploring the above questions, they should not be used uncritically. Ultimately, a persona is a simplified generalization that stands in for a group of people. This definition is almost identical to the definition of a stereotype, so it is unsurprising that persona-based design can reinforce stereotypes. There is a significant body of recent HCI research that explores how to minimize the risk and impact of stereotype-related bias in design processes that use personas, which should be used to guide the real-world application of personas.

Examples: Microsoft Personas

Microsoft has published a standard set of developer personas, which can be used to explore how different programmers engage with programming languages. We reproduce the Microsoft personas here for the purpose of discussion:

The Systematic Developer

Writes code defensively. Does everything they can to protect their code from unstable and untrustworthy processes running in parallel with their code. Develops a deep understanding of a technology before using it. Prides themselves on building elegant solutions.

The Pragmatic Developer

Writes code methodically. Develops a sufficient understanding of a technology to enable them to use it. Prides themselves on building robust applications.

The Opportunistic Developer

Writes code in an exploratory fashion. Develops a sufficient understanding of a technology to understand how it can solve a business problem. Prides themselves on solving business problems.

Each of these personas captures three facets of a programmer's psychology:

- * How much risk and uncertainty the programmer tolerates during the programming process
- * The depth of knowledge the programmer wishes to acquire about the specific technologies they use
- * What makes them proud of their work

The Microsoft personas are short. We consider additional, longer persona descriptions.

Examples: Personas by Aldrich & Coblenz

The educators Jonathan Aldrich and Michael Coblenz have published example personas for use in courses about programming language design. Their example personas are reproduced here for discussion.

Charlie

Charlie is in her late 20s to mid 30s. She has a Bachelor's degree but not necessarily in IT. She's a self-taught developer. Her coding is unconventional and she mixes genius lines with simple errors. She seeks to reinvent her software development career but the how is still unclear.

Charlie has a family, which makes financial stability and work-life balance essential. She's new to the industry and thus looks for a company that offers a supportive, people-oriented environment, where she can learn and improve her skills.

Robin

Robin is in his mid 20s and has completed his formal education, such as a Bachelor's degree in computer science. He is probably on his second or third job but has reached the ceiling in his current job, as in, he has learnt a lot and gained experience but would be keen on taking the next step to further his career. Even though he probably hasn't taken any steps to find a new job

(applied), he is on the lookout for something challenging as well as purposeful. In his current role, he can be found working in a specialized programming area (front-end/back-end/mobile). On a personal level, he is probably in a relationship, he is also quite introverted and self-aware. He enjoys working on complicated tasks and really wants to be involved and feel a part of the company. He values transparency and is happy working with inspiring leaders. He's keen to know what is going on and where the company is headed. Salary isn't his top priority (as long as it is not too far below average). Instead, Robin appreciates non-financial rewards, especially those that make him feel valued for his work.

Check your understanding: What facets of programmers' psychology (and their lives more generally) are discussed in the Charlie and Robin personas, which are not discussed in the Microsoft personas?

Answer: These personas discuss gender, age, educational background, career goals, and personal goals.

Comparing the Microsoft personas against Charlie and Robin, we see that different personas from different designers might address drastically different aspects of the programmer's life. The Microsoft personas are narrowly focused on three aspects of programmer psychology, while Charlie and Robin take a more

expansive view that places greater value on their demographic information and their life priorities.

When we choose whether to use a narrow or broad persona, there are potential design trade-offs. The narrow Microsoft personas focus on issues that have a clear and direct connection to the task of programming, which may encourage design discussions that focus on universal usability issues that affect all users. The limitation of these personas is that if demographic aspects of the persona (such as their education level or their personal goals) present unique usability needs that require unique solutions, the Microsoft personas may not lead designers to address those needs during design conversations. Conversely, though the Charlie and Robin personas invite the designer to confront these important non-universal design issues directly, this introduces the potential for a designer's social biases to influence the design discussion, an issue for which countermeasures should be taken.

Goal-Setting

Identifying the programmer's goals and the programming language's goals can be substantial tasks in their own right. This task has been well-studied in the context of Software Engineering, where it is known as requirements gathering. For extensive

coverage of this topic, we refer the reader to texts on requirements gathering.

For the purposes of this book, the key observation about goal-setting is that goals can be gathered from different people, such as:

- * Professional programmers
- * Students
- * The language designer

and that there are tradeoffs in the potential goals of each group:

- * Professional programmers are intensive users of programming languages, so the strength of consulting them is that addressing their needs would potentially have an outsized positive impact. A limitation of consulting professional programmers is that because they are immersed in the development of production code, there is a potential for anchoring bias: their stated goals may be closely tied to common programming tasks, and not reflect the full diversity of potential language designs.
- * Students are actively acquiring knowledge about the usage of programming language. A strength of consulting students is that because they have had less opportunity to acquire preconceived assumptions about programming languages,

they may able to identify and challenge limitations of previous languages. A limitation of consulting students are that in many cases their experience may be limited to writing smaller programs with limited maintenance needs (programming-in-the-small).

- * Language designers carry their own goals, whether the designers are professional programmers, students, or academic researchers. The limitations of designing a language based on the designer's own preferences are well-known: a language based on a single designer's preferences may not need the needs of a broader population. If these limitations are embraced honestly, however, there are also advantages to setting your own goals. In an educational setting, picking your own goals is valuable because it gives you agency in the design of a language, which provides motivation and thus promotes engagement in the learning process. In a research setting, embracing the right of the designer to pick their own goals broadens the space of designs they can pursue, enriching the set of design ideas that future designers might build on.

While working through this book, the reader is encouraged to identify their own goals as a designer by providing their own answers to the questions of (1) who the user is, (2) what tasks the user wishes to perform, and (3) in what context the user wishes to perform those tasks.

Example statements of goals are provided.

Example 1:

- * Who: Professional programmers working on low-level systems software, e.g., software which must interface directly with hardware or operating system functionality
- * Tasks: Eliminate memory errors such as double frees, use-after-frees, and memory leaks from their code
- * Context: The programmers are paid contributors of an open-source programming project with decentralized project planning among a globally-distributed mixture of paid and volunteer programmers of different experience levels

Example 2:

- * Who: Professional visual artists working in serial art formats such as comics or manga
- * Task: Automate the generation of dialogs and storyboards for use in serial art by using a program to express the world in which the art takes place
- * Context: Each of the artist's projects may have different funding sources and funding schemes, some of which are contract-based and others of which are commission-based. Clients may have highly specific requirements. In addition to

work-for-hire, the artist may produce other works as portfolio pieces to advertise their abilities

Example 3:

- * Who: Undergraduate computer science majors in their first year.
- * Task: Develop proficiency in specific core programming techniques such as iteration, Boolean logic, dividing a complex program into separate functions, and recursion.
- ntext: The student is programming in a controlled classroom setting. This provides substantial flexibility in the choice of programming language design. However, the student is motivated by the potential to continue programming in the future using popular, existing programming languages, and this motivation may influence their language preferences for classroom use.

Satisfaction

The ISO 9241-11 standard defined satisfaction as a key aspect of usability, that is, it places importance on what users like. Though the upside of pursuing a well-liked programming language design is self-evident, it is worth engaging with the limitations of assessing programmer satisfaction. Because the question of satisfaction is an inherently human and subjective question, it

must engage with the complexity of human decision-making, including human bias.

Several forms of user bias in satisfaction ratings have been studied. We mention two such forms of bias:

1. “We find that respondents are about 2.5x more likely to prefer a technological artifact they believe to be developed by the interviewer, even when the alternative is identical. “ [Dell et al., CHI 2012]
2. “When the interviewer is a foreign researcher requiring a translator, the bias towards the interviewer’s artifact increases to 5x.

Though the foreign researcher with an interpreter receives a **higher** rating in this study, it still introduces a unique challenge for interviewers who use interpreters, because it makes it harder for them to obtain high-quality data.

Another fundamental limitation of satisfaction ratings is that, in their simplest form, they are not actionable. If a user likes a program, this does not tell the designer what to do. If a user does not like the program, that also does not tell the designer what to do. This does not mean that user satisfaction should not be studied at all, rather that more precise instruments should be used to provide satisfaction feedback that is insightful and actionable.

This limitation of satisfaction ratings highlights a fundamental distinction between two ways that feedback collected from users can be employed in a design process: *formative* feedback is feedback used to inform future work, while *summative* feedback is feedback used to evaluate the success or failure of past work. A simple satisfaction rating can serve a *summative* purpose and determine whether a design achieved its desired level of programmer satisfaction, but it struggles to serve a *formative* purpose, it struggles to determine the direction of future design work.

HCI Methodologies and Paradigms

The discussion of usability in this chapter has highlighted how even once a question has been identified (such as satisfaction) it is important to pick an appropriate *methodology* for studying that question which provides information that is truly useful for the designer's specific needs. The following chapters will discuss common methodologies in greater detail; we provide an overview here. HCI methodologies used for programming language design broadly fall under two research paradigms: *Quantitative Methods* and *Qualitative Methods*.

Quantitative Methods

Quantitative methods have the following characteristics:

- * They emphasize quantitative (i.e., numeric) data
- * They emphasize the interpretation of those data through statistical methods
- * They often seek large, diverse samples of data to ensure the statistical significance of their conclusions

Qualitative methods

Qualitative methods have the following characteristics:

- * They emphasize the depth of data over the scale of data
- * They often emphasize data that cannot be reduced to numbers (though qualitative methods can incorporate numbers as well)

Computer scientists often find quantitative data appealing: after all, computers are great at crunching large data sets. This book emphasizes qualitative studies because they are good at addressing open-ended questions, e.g.:

- * What aspects of a programming language produce the most joy or excitement in the programmers?
- * What aspects of a programming language produce the most frustration or discouragement in the programmers?
- * How effectively did the design documentation for the programming language communicate the designer's intentions to the programmers?

Different qualitative methods vary in just how open-ended they are. On one end of the spectrum, an interviewer may collect an open-ended set of answers to a fixed set of interview questions. On the other end of the spectrum, an experimenter could give a programmer an open-ended list of tasks to perform with no prompting on how to complete them, then observe user behavior. Such studies are not controlled and not suitable for comparing competing tools, thus they are not classified as “experiments” in the strict sense, rather they excel at detecting *new* problems.

Generalizability

In both qualitative and quantitative studies, any study is only performed on a finite group of people, not the complete potential population of programmers. Thus, it is not guaranteed that insights about the test subjects will generalize to all potential

programmers. There are several approaches to mitigating thus, but none are perfect:

1. The people in your study should belong to the population you want to study.
2. If your sample population is similar enough to the audience population, you can try to argue that results generalize from one to the other.

Before attempting approach #2, it is important for the designer to interrogate whether the results are likely to generalize across populations and provide concrete justifications.

Classroom Activities

1. If the class incorporates usability studies performed by students, class time should be dedicated to the discussion of common issues in study design. Students should be made aware of any constraints on the length of a study. If students' user studies are used to drive their own project work, explore how to write actionable questions. If students have a particular programmer population in mind, explore whether that population matches the class population. If graduate students or other researchers are present, clarify that classroom studies are educational activities that are, by default, not publishable in research, and that appropriate approval (e.g., by an Institutional Review Board) would be

required for publication of study results. In a graduate course, discussion time can be dedicated to ethical issues in study design, such as the ethics of recruitment.

2. Working in small groups of 2 or 3, identify a usability goal together, i.e., identify answers to the six usability questions. This can be done in several ways:
 - a. Each person in the group answers 2 or 3 of the questions, then the group rotates, and each student builds on the prior student's answers when answering the next questions
 - b. Each person in the group answers all of the questions, then discusses their answers with a partner and jointly proposes a new set of answers
3. Myers-Briggs Type Indicators (MBTIs) are a popular system for describing personalities. Have each student draw MBTIs out of a bag. Write a user persona for a programmer with the given personality. Share the results.
4. Have each student pick from a set of personas. Give the students a design question to discuss. Have the students engage in a roleplay where they each discuss the design question as their persona would see it.

Exercises

1. Autoethnographic Essay: Read the personas Charlie and Robin, and note the aspects of their personal background which are described in the personas. Think back to the first programming language you learned or the language that has impacted you the most. Discuss how your background influenced your experience of that programming language.
2. Pick one of the aspects of personal identity discussed in Charlie or Robin and identify a usability question where there is potential for that aspect of identity to affect usability. For inspiration in choosing a usability question, consider consulting your own or others' lived experience, browsing online communities of programmers, or exploring a specific programming language feature that has been introduced in prior coursework.
3. Design a programmer persona. Your persona should include all of the aspects discussed in the Microsoft personas and at least one additional aspect discussed in Robin or Charlie. This persona could be based on any of the following:
 - a. Yourself
 - b. A public figure
 - c. A fictional character
4. If you are developing a programming language design project, answer the six usability questions to identify a

concrete usability question about your own design.

Related Work

This chapter is based on lectures by Jonathan Aldrich and Michael Coblenz and the following paper:

<https://faculty.washington.edu/ajko/papers/Myers2016ProgrammersAreUsers.pdf> Course URLs:

https://cseweb.ucsd.edu/~mcoblenz/teaching/291I_fall2022/Lecture3.pdf <https://www.cs.cmu.edu/~aldrich/courses/17-396/>

¶

The unreadable code example is a classic example from the source code of Quake III Arena.

(Source: Wikipedia, originally from Quake III Arena source code)

(ISO 9241-11, Ergonomics of human-system interaction)

¶

Example personas are provided from Microsoft

Cite work on avoiding stereotype issues in personas

Cite work on requirements gathering

Cite Chris work on comic generation

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter10

Chapter 11

Surveys

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * Surveys are a fundamental tool for collecting human feedback across the social sciences, including feedback from programmers about programming language designs
- * Surveys can incorporate qualitative data, quantitative data, or both
- * Some surveys include demographic data. We use this opportunity to discuss how surveys engage with data about human identity. Common issues of representing demographics in surveys are summarized in the Worst Survey Ever.

- * Psychometric survey questions such as Likert scales and Likert-type data are discussed
- * Accessibility issues in survey design are discussed
- * Acquiescence bias, one potential bias in the responses to survey questions, is explored.

Surveys are used throughout the social sciences and beyond to collect information from human subjects for various purposes, such as measuring the social condition, aggregating shared knowledge, measuring mood, or observing opinion. In design work, including but not limited to programming language design, surveys are used for purposes such as assessing people's assets, goals, needs, preferences, and their opinions on a proposed design.

Survey Data

Surveys are ultimately a mechanism for collecting data from humans, so we explore the different kinds of data that can be gathered by a survey.

Quantitative Data

Common quantitative (numeric) data collected in surveys include measures of the subject's experience level, performance metrics,

and measures of thinking (psychometric data). The following are examples of potential questions about experience level:

- * How many years of programming experience do you have?
- * How many hours of programming did you do per week in that period, on average?
- * How many years of formal programming education have you completed?

Examples of actionable quantitative performance metrics, potentially suitable as formative feedback, include:

- * What percentage of error messages you encountered provided information that led you to resolve the underlying error?
- * How many times did you consult the documentation page for a standard library function to obtain specific information, and of those times, how many times did the documentation page contain the desired information?

Examples of quantitative performance metrics which may not be actionable, but may be employed as summative feedback, include:

- * How many minutes did it take to complete the given programming task in the given language?

- * How many times (measured by how many times you invoked the compiler or interpreter) did you revise your program before it successfully implemented the given task?

In addition to measuring programmer experience levels and performance metrics, quantitative survey questions are often used to measure subjective experiences and opinions, typically through a style of scale called a psychometric scale. The best-known style of psychometric questions are Likert questions. A Likert question starts by providing a statement and asking the respondent how strongly they agree/disagree.

Example Likert question: How strongly do you agree or disagree with the statement “I find the programming language Python applicable to the majority of programming tasks I encounter”?

1. Strongly Disagree
2. Disagree
3. Neither Agree nor Disagree
4. Agree
5. Strongly Agree

Likert questions are useful because they convert subjective questions about human thought or experience into a numeric format for further analysis. Likert questions are broadly

categorized into two distinct types, which require different methods of analysis:

- * A Likert scale is a series of Likert questions which all aim to assess the same underlying variable, and thus should be analyzed as a collection.
- * Likert-type data are individualized Likert questions which attempt to assess different underlying variables, and thus should be analyzed separately.

As an example of how Likert scales and Likert-type data require different kinds of analysis: the recommended notion of “average” (central tendency metric) is different for each kind of data.

Computing a mean is only meaningful for Likert scales, not Likert-type data. If a central tendency metric is used at all for Likert-type data, a median or mode is typically recommended instead.

Forced Choice

Whenever you write a Likert-scale question, ask yourself: Do I want to let the respondent remain neutral or do I want to make them take a side?

Both options are valid and common. If you want them to take a side, that’s called forced choice. A 4-value forced-choice Likert scale looks like this:

1. Strongly Disagree

2. Disagree

3. Agree

4. Strongly Agree

Acquiescence Bias

Several cognitive biases can occur a participant's responses to Likert-scale questions. One such bias is called "acquiescence bias": some groups of respondents have a bias toward agreeing with whatever statement they are given. A standard technique for measuring acquiescence bias is to divide the questions equally into *positive-keyed* (PK) questions, where agreement corresponds to a higher value of the underlying variable you wish to assess, and *negative-keyed* (NK) questions, where *disagreement* corresponds to a higher level of the underlying variable.

As an example, suppose we wish to develop a six-question Likert scale to assess the underlying variable of "positive programmer sentiment toward the programming language Python". To account for acquiescence bias, we would develop three positive-keyed questions and three negative-keyed questions.

Examples of positive-keyed statements are:

- * “The thought of programming in Python gets me out of bed in the morning”
- * “Python is my most favorite language”
- * “Python brings me life satisfaction”

Examples of negative-keyed statements are:

- * “Python brings me misery”
- * “The thought of programming in Python keeps me up at night”
- * “Python is my least favorite language”

To compute the acquiescence bias of a respondent’s answers, we compute the sum of all scores, both positive-keyed and negative-keyed. If this metric is greater than the average score of all the answers, it indicates a bias in favor of agreement. If this metric is less than the average score of all the answers, it indicates a bias in favor of disagreement. To compute an estimate of the underlying variable, we subtract the average score of negative-keyed answers from the average score of positive-keyed answers

Qualitative Data

Qualitative methods beyond surveys are discussed in a separate chapter. This section focuses more narrowly on the use of

qualitative questions within surveys.

In developing qualitative questions for surveys, it is worth noting that constructing a response to a qualitative question typically takes more time and effort than responding to a quantitative question. In contrast to Likert scales, where multiple similar questions may be asked to assess the same underlying variable, qualitative questions should avoid duplicating one another, in order to conserve the time and energy of the respondent.

When in doubt about whether to include a given qualitative question, revisit the precise usability question your survey seeks to assess, and only include the qualitative question if it is relevant to the overall goal of the survey.

Examples of Qualitative Survey Questions:

- * “What are the tradeoffs if my language is implemented as a compiler vs. interpreter?”
- * “How should we balance our resources on better error messages vs. a more complete standard library?”

Open-Ended Language

To get extensive, open-ended responses, questions should be phrased in an open-ended way, using words such as “How” and

“Why” that invite a respondent to give their full thoughts, rather than a short yes/no answer. Yes/no data

have their place, but they are quantitative.

At the same time, the level of open-endedness should be adjusted based on how much potential remains to adjust the design of your language. In early stages of development, it is appropriate to solicit feedback about major, fundamental design decisions, but as implementation progresses and commitments are made to particular design decisions, the questions asked typically narrow.

Demographic Data

It is common for surveys to collect data about people, specifically the respondents. In order to respect the rights of respondents, including the right to privacy, special attention should be paid to demographic data in survey design.

Justification

One of the fundamental design principles for the treatment of demographic data in surveys is that when you ask the respondent for a piece of information about them, you should be able to provide a clear and convincing reason for requesting that

information. We provide examples of clear reasons for requesting specific demographic data:

- * “Please provide your email address so the researchers can contact you to send you payment for participation and advertise follow-up studies”
- * “Please provide how many years of programming experience you have, so that we can compare experiences of new and experienced programmers”
- * “Provide your race and gender so that the research team can assess the effectiveness of their outreach and recruiting approach”

Self-Identification

A second major design principle for demographic information in surveys is that when demographic information such as gender, race, or personal pronouns are requested, they should be open-ended questions which allow the respondent to self-describe their identity. That is, these questions should be asked as fill-in-the-blank questions, not questions where the respondent selects a single answer from a fixed list.

In rare cases, there are compelling reasons to use a fixed list of options for race or gender. The designer who interprets the survey results might have need to compare their data against existing

datasets which use limited categories, such as fixed racial and gender categories in government census records. For the study of programming languages, however, the need to interface with such records is likely rare.

Simplifying the analysis of data is not a compelling reason to limit the allowed data values. If one needs to report simplified data that combine equivalent or near-equivalent values (such as “female”, “F”, and “woman” for a gender field), this simplification can be performed as a data processing step after the data are collected.

Accessibility

To enable collecting data from the full range of potential participants, it is important to design surveys in a way that is accessible to disabled participants. The particular designs required will depend on the nature of a person’s disability and how the survey is delivered, naming in person or online:

Visual Disability

- * For printed surveys where participants may have limited vision, a large-print version of the survey should be made available

- * For participants with no vision, ensure that digital surveys are compatible with screen-readers or, if the survey is in-person, ensure an alternative to the written survey, such as an interview, is provided
- * If color is used in the survey to communicate information to the reader (e.g. in figures), it should not be the only way that information is communicated, so that colorblind participants can receive the same information

Neurodevelopmental Disability

- * For printed surveys with dyslexic participants, dyslexia-friendly fonts should be used. Comic-style fonts such as MS Comic Sans are often used for this purpose
- * Participants with attention issues should be permitted to take breaks
- * In-person studies whose participants have sensory issues should make sure to choose a location that does not aggravate those issues (e.g., a quiet place with no bright lights and no strong smells)

Physical (e.g. Musculoskeletal) Disability

- * Some people cannot write or can only write for limited periods of time, using special arthritis-friendly writing tools. Providing digital surveys enables such people to participate. If in-person participation is required, advance notice should be given that writing is required, so that participants can bring appropriate tools.
- * If you are preparing a physical space for participants to take the study, make sure there is proper furniture if the participant needs to sit. Avoid places that only have stools with no back.

Worst Survey Ever

The following fictional survey intentionally makes as many design mistakes as possible, serving as a case study in worst-practice as a means of inspiring the reader to best-practice.

Content Notice: This survey represents personal data in a way that could be construed as disrespectful to marginalized groups, in an explicit effort to discourage that practice.

Demographics

Legal Name:

Preferred Name:

If you have undergone a legal name change:

Previous name:

Date of name change:

Method of name change:

Location of name change:

Gender [select one]: Female / Male / Not declared

Gender Identity [select one]:

- * Agender
- * All genders
- * Androgynous
- * Bi-gender
- * Demi-female
- * Demi-male
- * Female
- * Gender Fluid

- * Genderqueer
- * Intersex
- * Male
- * Non-binary/non-conforming
- * Transgender
- * Transgender Female
- * Transgender Male
- * Tri-gender

Pronoun [select at most one]:

- * He/Him/His
- * She/Her/Hers
- * They/Them/Their
- * Xe/Xem/Xyr
- * Ze/Hir/Hirs

Disability Status[select at most one]:

- * Hearing Impairment

* Learning Impairment

* Mobility Impairment

* Speech Impairment

* Visual Impairment

Marital status [select one]:

* Single (United States of America)

* Divorced (United States of America)

* Partnered (United States of America)

* Married (United States of America)

* Separated (United States of America)

* Widowed (United States of America)

Race [select all that apply]:

* American Indian or Alaska Native

* Asian

* Black or African American

* Hawaiian or Other Pacific Islander

* White

Ethnicity[checkbox]: Hispanic/Latino?

US Citizenship Status[select one]:

- * US Citizen
- * Permanent Resident
- * Non-Citizen

Quantitative Questions

What year did you start using a computer?

What brand of computer was it?

How many minutes a day do you spend on a computer?

On a scale of 1-10, how proficient of a typist are you?

Enter your typing speed in characters per hour

Rate your responses to the following statements:

“I like the proposed language”

1. Strongly Disagree
2. Disagree

3. No opinion

4. Agree

5. Strong Agree

“I am likely to recommend the language to a friend”

1. Agree Lots

2. Agree

3. Neutral

4. Disagree Some

5. Disagree More

“I think more people should use the language”

1. Agree as much as possible

2. Disagree a bit

3. Agree

4. Neither agree nor disagree

5. Disagree More

“This language should come pre-installed on computers”

1. Agree as much as possible

2. Disagree a bit
3. Simultaneously agree and disagree but in equal amounts
4. Agree
5. Neither agree nor disagree
6. Disagree More

Check all the following boxes that apply:

- * This language reminds me of Java
- * This language has useful features that Java does not
- * This language could be useful to a programmer that struggled to learn Java

Qualitative Questions

“If time and money were no constraint and you could make any IDE for programming in this language, what are all the features it would have?”

“What is your favorite thing about the language?”

“Did your program feel fast?”

Classroom Activities

- * Hand out copies of the Worst Survey Ever to students. Give them time to read the survey and identify the problems in it. As a group, discuss those problems.

Exercises

1. Implement basic functions for interpreting quantitative survey data, such as mean, mode, median, and variance
2. Research an established Likert scale from a field other than programming language design, preferably one which comes with explicit instructions for scoring and interpreting the results. Implement a program which defines the results to the survey as a data type, scores those results, and displays the interpretation of the score. An example of a suitable scale for this exercise is the PHQ-9 questionnaire, which is used for initial clinical screening of depression symptoms

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter11

Chapter 12

Qualitative Studies

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * Compared to surveys, interviews provide a more flexible format for collecting qualitative data, where the interviewer can adjust their questions to the needs of each subject.
- * Observational studies can be used to understand how programmers would undertake a hypothetical task in a proposed language
- * Grounded theory and thematic analysis are techniques for rigorously interpreting the data collected in qualitative studies
- * Autoethnography is the scholarly technique of writing from lived experience, which can complement other qualitative

methods by providing depth on the experience of an individual

This chapter explores qualitative methods for studying human interactions with programming languages, beyond qualitative questions asked in surveys. Common qualitative studies include interviews and observational studies. Potential methodologies for interpreting the resulting data include grounded theory and thematic analysis. We also discuss other qualitative techniques such as autoethnography which are rarely used by name in the study of programming languages, but may be used without knowing it.

Interviews

Interviews are widely used to collect qualitative data from subjects, especially if the investigator desires greater flexibility than a survey provides. This flexibility includes the ability to provide and request clarification, tailor follow-up questions for each subject, explore unexpected topics, and make use of nonverbal communication. The cost of achieving this flexibility is that interviews typically require a greater time investment from the investigator than surveys do, especially at scale.

When planning to perform an interview, prepare a list of interview questions in advance. Interviews are often semi-structured, meaning that although you prepare a specific structure, you are not restricted to it. You want to learn the person's experiences and thoughts, which may take you off-script.

Writing your interview script

Your script should start by obtaining consent to record the interview. Next, it should start with the most general questions, proceeding to more specific ones later on.

When writing questions, include follow-up questions. A particularly important kind of follow-up questions are probe questions, e.g.

- * “How do you mean that?”
- * “Tell me more about that”
- * “Anything else?”

These probe questions significantly increase the number of responses, often receiving more detailed responses than the initial question.

Questions should be phrased in a neutral tone, not implying that there is a single correct answer.

Observational Studies

If you want to see how a user interacts with your language, there's no replacement for simply letting them interact with it and observing. This style of usability study is open-ended and often exploratory, helping identify new problems that you couldn't predict. Because of their open-ended nature, they're not "experiments" in the strict sense, but certainly are research.

Picking Tasks

This is perhaps the hardest part: what tasks will participants do? You will get this wrong the first time, and the only solution is to iterate and revise your task choices. For this reason, it is advisable to test tasks with an individual or small group before performing a full study.

Example Starting Points:

- * “Write a program that satisfies this specification”
- * “Fill in the missing code to satisfy the specification”
- * “Read this code. Are there bugs? If so, which?”

- * “Here is a debugger. Debug this code”
- * “This code does not compile. Modify it so that it compiles”
- * Parsons problems: Given these code snippets, put them in right order.

Preparation

- * Identify whether your participants need training from you before study. If so, be wary of using only written materials. People may not read the materials, or may incorrectly assume they fully understood the contents. If there is training, you should assess whether the subjects successfully learned what the training sought to teach them.
- * Decide which tools will be used, such as paper/pencil, text editors, compilers, debuggers, and/or test frameworks. If using paper/pencil, be prepared as the interviewer to simulate any software that have not been implemented yet.
- * Know how much information and help you’re willing to provide, in very clear terms, before you start.
- * Rehearse interviewing best-practices such as using probe questions, asking one question at a time, speaking in clear, simple terms, and providing adequate time to answer.
- * Bring a notetaking device in addition to any recordings, so that you can go through the recordings “as-needed” rather than in their entirety.

- * Keep a timer/clock in view so that you can record timestamps in your notes

Revising Observational Study Tasks

- * Identify the most important point of your language design, design task for that
- * Task should be easy enough to be possible, hard enough to be meaningful
- * Don't give too many tasks
- * Minimize distractions, e.g., you don't want a subject to spend 30 minutes exploring whitespace questions if the language is whitespace-insensitive.
- * Narrow task scope to only things you care about
- * For big monolithic tasks, consider breaking them down into small tasks (unless your point is to assess integration of the smaller parts, big-picture thinking, etc.)

Collecting Data

When observing a participant perform a task, there are multiple ways to collect data:

- * Audio + Video + Screen recordings

- * Eye tracking (expensive!!)
- * Post-study Surveys (see Survey lecture)
- * Think-aloud: have them think through thoughts out loud either after or during study. Prompt them to keep talking.
- * Take lots of notes

Interpreting Qualitative Data

Motivation: If we don't want our preconceived hypotheses to guide the research too much, then let's just... not develop hypotheses in advance.

Instead, observation of the study subject comes first. Record events that occur, then perform “coding,” a process of assigning attributes to each data point, not to be confused with “coding” in the sense of computer programming. The exact process of coding differs depending on which methodology (such as grounded theory or thematic analysis) is used to interpret the data.

In a formal grounded theory process , coding consists of three phases, respectively called “open”, “axial”, and “selective” coding. Our description of these phases is based on an exposition by Tiffany Gallicano.

To do open coding, read through your data several times. Don't use an existing theory to assess the data, just look for patterns that emerge from these data, to come up with tentative labels for the data.

Example (from Gallicano):

“Research question three: What irritates or upsets Millennials when receiving feedback on their work?”

Open codes:

- * Getting called out
- * Not being heard
- * Mind reading and miracle-worker expectations

For each code, the coder (the person who performs the coding process) lists example of participants own words that were assigned that code, and recurring patterns/themes/experiences for each code.

Next, to establish relationships among codes, generate “axial codes” which each apply to some set of the open codes.

For example, “Not being heard” and “mind-reading” could both be axially coded as “communication failures” and “getting called out”

might be axially coded as “public shaming”.

Lastly, selective coding is the process of putting categories into categories (core categories). Perhaps a selective code that combines “communication failures” and “public shaming” would be “absence of nurturing communication style.”

Additional Qualitative Methods

The following methods are less often used by name in the study of programming languages, but see substantial usage in other fields.

Ethnography

In an ethnographic study, the researcher spends a significant period living among a population and writes about their experiences, often in a long-form format. In traditional ethnography, a researcher from outside a community speaks as an authority on the experiences of that community. This outsider authority leads a major limitation of ethnographic studies: because outsider experiences may struggle to capture the full depth of a community’s experience, treating that experience as authoritative carries the potential to propagate an inaccurate understanding of the community.

The following methods are often employed in an effort to overcome the limitations of ethnography and ensure cultural authenticity of results.

Research-Practice Partnership

In a research-practice partnership, the outsider researcher takes on the “subjects” as proper partners, meaning that they get to participate equally in steps ranging from the identification of research questions to writing and publication, typically resulting in shared credit on any academic research publications.

Example: A programming languages researcher partners with the non-profit community organization that oversees the direction of future development for a particular programming language.

Together, they determine which questions about their programming community are of mutual interest, and investigate those questions together.

Autoethnography

Autoethnography is when a researcher applies ethnography to themselves. Instead of overcoming the ethnographic power dynamic through teamwork, it is overcome by building a research team consisting of the subject group. Like all methods, it has limitations:

if a sole author or internally homogenous group of authors attempt to speak on behalf of an entire identity group, they will fail to fully represent that identity group.

An argument in favor of autoethnography would be to recognize that no sole method can fully represent issues of identity, and that research fields make their progress through the combination of methods by different researchers.

Example: In 2020, a team of Black HCI researchers wrote autoethnographically about their lived experiences of racism within HCI.

Example: Prof. Zoe Reidinger at WPI has written authethnographically about queer issues.

Example: When a programmer writes a social media post describing why they like or dislike a particular programming language, they are engaging in autoethnography

Because authoethnography means researchers writing about ourselves, it risks overemphasizing the importance of issues faced by researchers rather than the general public, yet it does excel at addressing those issues.

Concluding Remarks

This chapter has emphasized the division of methodologies into quantitative vs. qualitative. It is also helpful to divide studies by role: formative vs. summative. A formative study helps guide the direction of research before the research is finished; A summative study helps assess research's success or failure at its conclusion.

Both of these roles are important, but formative feedback is particularly valuable in the classroom, where revision and growth are important.

Related Work

<https://prpost.wordpress.com/2013/07/22/an-example-of-how-to-perform-open-coding-axial-coding-and-selective-coding/>

Classroom Activities

- * Hand out interview scripts to students and have them practice interviewing one another in small groups
- * Hand out an autoethnographic article about a programming language to students and have them perform open, axial, and selective coding.

Exercises

- * Take the question “What about C++ is better than Java?” and:
 - * rewrite it in a neutral tone
 - * write your own follow-up question
 - * write a probe to go with that question
- * Collect three articles (or posts) that describe personal experiences with the same programming language. Perform open, axial, and selective coding on these articles

Appendix: Example interview script (from Aldrich)

1. How long have you been programming professionally?
2. Can you give an order of magnitude estimate of the size of the largest project you've made significant contributions on?
Number of people, lines of code?
3. In what programming languages do you consider yourself proficient?
4. How did you get into software development? Do you have a computer science background?
5. Let's talk about changes that happen to state in software you've worked on. Many kinds of software maintain state,

such as object graphs, files, or databases, but there's a possibility of corruption during changes due to bugs. How do you make sure that state in running programs remains valid?

6. Are there specific techniques do you use? If so, what are they?
7. Do you sometimes want to make sure that some operations don't change any state or don't change certain state?
8. Tell me about a recent time you did this.
9. How often does this come up?
10. Do you use language features to help?
11. Do you sometimes want to make sure that some state never changes?
12. Tell me about a recent time you did this.
13. How often does this come up?
14. Do you use language features to help?
15. Do you sometimes want to make certain kinds of modifications to state impossible for some users of an API but not others? If so, how do you do that?
16. How often do you work on concurrent aspects of software? What mechanisms do you use to control concurrency?
17. Do you use immutability to help address or prevent concurrency issues?

18. How much work have you done on security-related aspects of your software? Have you found or fixed any vulnerabilities?
19. Do you use immutability to help address or prevent security issues?
20. Can you recall a recent situation in which you created an immutable class or other data? If so, tell me about it.
21. Can you recall a recent situation where you changed a class from immutable to mutable? If so, tell me about it.
22. Can you recall a recent situation in which you changed a class from mutable to immutable. If so, tell me about it.
23. Can you think of a bug you investigated or fixed that was caused by a data structure changing when it should not have? What was the problem and how did you solve it (if you solved it)?
24. Would const have prevented the bug?
25. Have you ever tried using an immutable class and had it not work? Why not?
26. When you create a new class, how do you decide whether instances should be immutable?
27. Have you ever been in a situation where you wanted to use const or final but it didn't say what you wanted to say?
28. or where you discovered you couldn't use it? What was the situation and why couldn't you use it?

29. Have you been in a situation where you had to revise your plan because something you'd assumed could mutate state was disallowed from doing so due to const?
30. Have you been involved in training new members of the team? What do you tell new members about immutability or ensuring invariants are maintained?
31. Sometimes, though an object is mutable after creation, it only needs to be changed for a short amount of time. For example, when creating a circular data structure, the cycle must be created after allocating all the elements. After that, however, the data structure doesn't need to be changed. Have you encountered situations like that? Do you think it would help if you could lock the object after all necessary changes were made?
32. Now, I'd like to move on to API design in general. Think of a recent API you designed.
33. Did you make any conscious design or implementation decisions, to make the API easier or more manageable for these users?
34. Are there any recurring issues / challenges that users have had with your API? How did you handle those?
35. How do you differentiate between users of your API? Are there parts of the API that you expose some users but not others? How do you manage that?

36. Did you make any conscious design or implementation decisions to protect key data or data structures from modification (inadvertent or malicious) from your users?

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter18

Chapter 13

Gender

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * The Humanist can use programming languages as a lens to deconstruct the social values underlying computing, including values associated to genders
- * The relationship of programming language concepts such as types with gender can be explored using the concept of structuralism from philosophy
- * The Social Scientist can use gender as a lens to explore the diversity of interaction styles and inform the interaction styles designed into a language

What does gender have to do with programming languages? In one sense, it can be argued that these subjects are unrelated: when a programming language is viewed as a pure mathematical abstraction, it is implicitly gender-neutral. The mathematics underlying a programming language can be understood as a passive thing which exists separate from society, and which can freely be used by any person who wishes. In making this argument, one might appeal to the global nature of the programming languages community, and the ability for programmers throughout the world to self-study even without access to formal education.

This chapter explores the opposite assertion, that the study of programming languages and the study of gender can be connected to one another in a mutually beneficial way. We explore these connections both through humanistic methods, focusing on the philosophical concept of structuralism, and through social science about gendered experiences of computing.

Structuralism

Structuralism is one of the major paradigms in philosophy, historically prior to the paradigm of post-structuralism. Because these two paradigms each encompass a broad array of work across decades of time and across many disciplines, they defy

concise definition, so this chapter focuses on one specific difference between them: universalism.

Structuralism tends to seek universal descriptions and analyses of the world, which apply uniformly across it. As an example from moral philosophy, Kant defined morality using the *categorial imperative*:



Act only in accordance with that maxim through which you can at the same time will that it become a universal law

-- Immanuel Kant

Kant's concept of categorical imperative makes an explicit universalist claim, that only universal laws are grounds for morality. Across other areas of philosophy, structuralist thinkers have divided the world into clearly separated rules and proposed universal rules about those groups. For example, the philosopher of economics Marx proposed material dialectics as a universal principle for understanding economic relations between clearly-delineated groups: workers and owners.

Structuralist thinking about programming languages falls out of structuralist philosophy through a clear and direct link: formal

logic. The theory of programming languages is inseparable from the study of logic, which simultaneously belongs to mathematics and philosophy. Major logicians from history serve as an explicit link: as an example, the logician Frege has Kant as an influence, and Frege in turn inspired the same school of logicians which use logic to reason rigorously about the correctness of software.

Programming language theory, specifically type theory, is arguably the most fruitful modern application of structuralist thinking. Type systems are exceptionally well-suited to structuralist thinking, because dividing data and code into clearly-defined interacting components is a fundamental part of most Practitioners' programming processes. For the Practitioner, the act of defining their world carefully gives them a pay-off: more flexible and robust code. When the Theorist thinks in terms of universal properties, it is exceptionally productive: the Practitioner and Implementer benefit directly when the Theorist guarantees that *every* well-typed program is well-behaved or proves that a program satisfies its specification in *every* case. In the context of programming language theory, structuralist, universalist thinking is well-matched to expressed needs of the humans who use programming languages. Programmers benefit from knowing that when expression `e` has type `int`, its value is an integer. They benefit from knowing type-safe programs don't

produce memory errors, that garbage-collected programs do not contain memory-leaks, and so on for countless other theorems.

Post-structuralism

The post-structuralist paradigm in philosophy emerged after structuralism, in direct opposition to it. Because the uniting theme of post-structuralism is that it challenges structuralism, it also challenges dividing the world in cleanly-divided groups and challenges universal claims about those groups, instead arguing that multiple truths exist at the same time. These multiple truths can be explained by different underlying assumptions, different priorities, or different lived experiences.

Post-structuralist philosophy often deals with people, and highlights how claims about one group in the world may not be applicable to others. As an example, the moral philosophy of Kant, who operated from a colonial center, has been thoroughly criticized by decolonial philosophers. Traditional systems of philosophy about the law have been criticized, e.g., by Foucault's explorations of the effects of surveillance on the experience of the surveilled.

Just as programming languages are a prominent application of structuralist thinking, contemporary gender studies is a

prominent application of post-structuralist thinking. Third-wave feminist theory highlights the concept that gender is socially constructed, not universal, and thus subject to evolution. On one hand, the social construction of gender highlights the notion that gender inequality is not a fundamental facet of the world, but one which was chosen by society, implying that the choice could be reversed.

Though applicable to all women, the notion of social construction has particular importance in transfeminist theory. Social construction of gender invites the ideas of gender autonomy and gender anarchy: humans can choose to change the meaning of gender, eliminate gendered power hierarchies, and embody genders beyond the binary of female and male.

This post-structuralist approach does not claim gender is non-existent, but rather highlights its sources. For example, Judith Butler highlights the role of performance in forming, Andrea Long Chu highlights the role of oppression in forming gender, and others highlight the role of self-identification in forming gender. Black feminists have highlighted the intersecting role of race in determining the formation of gender.

Combining Type and Gender

Types and gender, at their most basic level, describe groups. Types are characterized by sets of values, operations on those values, and universal properties of the values and operations. A gender is characterized as a set of people, and indeed the word “gender” shares its etymological root with the French word for a group.

Once we assert that types and genders both correspond to groups, we can begin to play with these two notions of group. We play with these notions by taking concepts from one and applying it to the other.

From Types to Genders

We propose these topics from type theory as potential sites of gender play:

1. An intersection type $t_1 \wedge t_2$ describes a single expression that simultaneously has type t_1 and type t_2 . How does the concept of intersection type compare with the notion of intersectionality in gender? For example, intersection types can be used to model object-oriented programs and subtyping. What insights or questions would result from comparing them against concepts such as objectification and tokenization?

2. Choice is a common feature of type systems, e.g., through ADTs. Some type systems such as linear types divide choice further into internal choice $t_1 \oplus t_2$ and external choice $t_1 \& t_2$. For an internal choice, the implementer chooses whether to implement t_1 vs t_2 ; for an external choice, the client chooses whether they use a t_1 vs t_2 . How do these competing notions of choice compare the notion of gender autonomy?
3. Type-based testing tools like QuickCheck generate sets of test data that reflect the diversity among values of a given type. Could a counterpart be developed which ensures designers explore the full impact of gender on software?

From Genders to Types

We propose these topics from gender studies as potential sites of play with types

1. The set of people with a given gender changes as people are born, live, and die. How does one develop a type system where a type can gain new values over time? How does one develop a type system where a type can lose values over time?
2. Not only does the set of people with a given gender change, but the set of genders that exist can change throughout the lives of individuals. How does one develop a static type system where the set of types can evolve throughout program execution?

3. How does the notion of performance in gender translate to the theory of programming languages? Can analogies be built between gender performance and the actor model of programming? Are programs better-suited to the notion of performance from gender studies, or the notions of performative and non-performative utterances from the philosophy of language?

The above questions are appropriate project exercises for Humanist students. Some of them have established answers as of this writing, others are open questions.

Structuralist Gender in HCI

This section explores an answer to the following question posed earlier in the chapter: “Type-based testing tools like QuickCheck generate sets of test data that reflect the diversity among values of a given type. Could a counterpart be developed which ensures designers explore the full impact of gender on software?”

We answer in the affirmative: HCI researchers have developed methods for systematically potentially exploring the role of gender in a user’s experience of software, methods which could potentially be applied to the design of programming languages. To achieve this goal, the researchers engaged with a structuralist understanding of gender. In this section, we both explore those

methods and criticize them through a post-structuralist lens. Specifically, we explore the GenderMag method and its descendants, such as InclusiveMag.

The core steps of the GenderMag approach are as follows

- * Identify aspects (facets) of a user which academic literature indicates differ by gender, on average
- * Develop a set of user personas that reflect diverse values of those facets
- * Employ those user personas in the remaining stages of your design process

We describe how GenderMag fits into broader research trends, describe facets, and describe extensions to GenderMag.

Empirical vs. Analytical Methods

The GenderMag method is an analytical method, which can be contrasted with empirical methods.

An *empirical* approach is one driven by data, typically data about users. Recruiting diverse users for user studies is key to inclusivity for empirical approaches. Researchers may cite this as a challenge in some cases, particularly when researching intersectional

populations with relatively few members. Datasets for intersectional populations are thus a key area of empirical work; a notable example is Buolamwini's dataset for computer vision.

An *analytical* approach (not to be confused with analytic philosophy) is an approach that is not driven by user data, but is focused on what the designers do on their own, in isolation from the user. Analytical approaches avoid the challenge of scarce participant time, but have significant problems of their own. With an analytical approach, a designer fundamentally does not get outside their own head, limiting the perspective from which they view their design. GenderMag is an *analytical* method, and its design centers on the relationship between dimensions of identities and facets of user personas.

Dimensions vs. Facets

A *dimension* is just any aspect of a person's identity that we might label, examples: race, gender, gender modality, money, sexuality, disability, neurotype, native language.

Facets are the type system of dimensions proposed by GenderMag. Importantly, dimensions do not correspond one-to-one with facets; facets are abstract features of a person which are more directly tied in to their computing interactions than their

demographics are. The example facets from GenderMag are: “Motivation”, “Computer Self-Efficacy”, ”Attitude Toward Risk“, ”Information Processing Style“, and “Learning by Process vs. Tinkering”.

The strength of facets lies in the strength of types, as abstractions. Just as a type system lays out the variety of data a program could need, these facets lay out the variety of usability concerns. At their best, facets could encourage a designer to ignore irrelevant details of a person in the same way that types encourage us to ignore irrelevant details of a value. These facets can guide design discussions for programming languages. We provide an example for each facet.

Motivation

This facet captures the range of motivations a programmer has for writing a program in a given programming language. At one end, programmers can have extrinsic motivations, writing programs because of what those programs can accomplish. At the other end, programmers can have intrinsic motivations, writing programs because of the joy of using a particular programming language in its own right.

To develop a programming language that includes both motivation styles, the programming language designers might

simultaneously ask:

- * What novel core language features are provided, which provide new opportunities for intrinsic enjoyment?
- * How do language designers create a language ecosystem which supports extrinsic enjoyment, e.g., by developing instructional materials featuring concrete practical applications or by developing libraries for major classes of applications?

Information Processing Style

Information-processing styles range from comprehensive processing styles, where a person attempts to achieve complete knowledge before proceeding with a task, vs. selective information processing, where a person proceeds in a task based on the first promising information, and backtracks to try a new approach. Information-processing style is relevant to both the core technical design of a programming language and the design of tooling:

- * Static type systems arguably reflect a comprehensive processing style: the programming language implementation itself comprehensively explores the behavior of every branch of a program, before executing any part of the program. Dynamic type systems reflect selective information processing, where execution proceeds without

comprehensive program analysis. If an error occurs in one branch of a program at runtime, it is handled selectively at runtime.

- * Developer documentation, including documentation, represents a sizeable collection of information which a programmer might process. Designers of documentation browsers could explore supporting both processing styles by allowing programs to either select detailed information on an individual function or object, or collect comprehensive information about a part of a library.

Computer Self-Efficacy

Self-efficacy does not mean competence, instead it means a specific kind of confidence, the confidence in one's ability to successfully complete a given task. Awareness of confidence is important to programming language design because confidence informs how programmers might react to obstacles, and the process of programming involves overcoming many obstacles

- * Programming languages and their implementations differ in how early and frequently they present error messages to a programmer. In general, statically-typed languages present more compile-time error messages to programmers than dynamically-typed languages, in the hope of preventing runtime errors. If errors become obstacles, then frequent

error messages have the potential of discouraging programmers who have low self-efficacy

- * Language implementations differ in their error messages, and thus in how they present the same underlying message to a programmer. Error messages can be engineered to provide concrete recommendations for how to resolve them. Beyond helping all programmers improve their programs, this has the potential to promote confidence. Language choice can also be modified to promote encouraging language, emphasizing that errors can be fixed, rather than their existence.

Attitude Toward Risk

Attitudes toward risk range from risk-averse to risk-seeking.

Attitudes toward risk can inform discussion of typing discipline and choice of languages and tools:

- * If runtime errors are understood as risks, then dynamically-typed languages carry additional risk relative to statically-typed languages.
- * The choice of tools can be understood as a social risk. If a new programming language with a small user base is chosen for a new software project, there is potential risk that it would lack necessary community support, libraries, or tools. For a student, learning less popular programming languages could be viewed as a career risk. When developing new programming language features, these risks could be

mitigated by integrating new features in existing languages or tools when possible.

Learning by Process vs. Tinkering

Preferred learning styles differ among programmers: some learn by tinkering, i.e., playing and experimenting with given features, while others learn by systematic process. Learning styles can be supported through compiler errors and through educational materials

- * Tinkerers can be supported by tooling that supports immediate execution of code, such as interactive read-eval-print loops (REPLs).
- * Systematic learners can be supported by providing exhaustive educational resources, and by linking individual error messages back to systematic concepts.

Extension: InclusiveMag

One self-evident limitation of GenderMag is that it only considers inclusiveness related to gender (which it views through a limited, binary lens). Inclusive software design, including inclusive programming language design, may face different issues when addressed to different *intersectional populations* of people who belong to multiple marginalized groups at the same time.

InclusiveMag is an extension of GenderMag intended to address inclusivity issues that disproportionately affect intersectional populations. A motivating example for InclusiveMag was the relative ineffectiveness of many computer vision systems for dark-skinned women, but one can imagine applying the same approach to programming languages as well.

The designers of InclusiveMag problematize inclusive design for intersectional populations as a matter of combinatorial explosion: if there are N dimensions to our identity and if each dimension had only 2 values, there would already be 2^n different combinations to test. This exponential growth in identity groups makes fully-inclusive design a legitimate challenge, though the details of the challenge depend based on which style of design methods one wishes to apply.

InclusiveMag, like GenderMag, takes a structuralist approach to representing human identity across dimensions such as gender, race, and disability. All dimensions and facets are conceptualized as one-dimensional, finite spectra, e.g., equivalent to the closed real interval $[0,1]$. Going a step further, InclusiveMag only uses the extremal values 0 and 1 for design, and throws out all intermediate values in the open interval $(0,1)$.

The basic technique of the InclusiveMag approach is to generate design personas by combining different sets of extremes $(0,1)$

from the different facets. The selling point is that if the set of facets remains limited, then even if the number of dimensions were to grow large, the technique provides a dimension reduction that keeps the total number of personas tractable. In InclusiveMag, the number of design personas only grows based on the number of facets, not the number of identity dimensions.

Criticisms and Limitations

The GenderMag family, including InclusiveMag, can be criticized from multiple perspectives. We present criticisms involving othering, reductionism, identity spectra, and intersectionality.

Reductionism

Fundamentally, GenderMag relies on facets as an abstraction for dimensions of identity. Programming languages teach us that the power of an abstraction depends on whether it is faithful, i.e., whether the assumptions we make using an abstraction hold true in practice. Abstractions achieve their power in large part by ignoring the details of the thing they abstract. In studying humans, in contrast to programs, there is greater risk that an abstraction erases an important aspect of a person's lived experience, i.e., there is risk of reductionism when we study experience through facets, particularly because the GenderMag facets only consider problem-solving styles.

Identity Spectra

Many of the identity dimensions explored in inclusive design, such as race, gender, and disability, are spectra, not binary.

Though InclusiveMag acknowledges these spectra, it erases them from the design process, focusing only on the extremes. Moreover, the spectra are represented as lines from one extreme to the other, ignoring their full complexity

Intersectionality

InclusiveMag advertises itself as an intersectional approach because it addresses intersectional populations: people who are marginalized along multiple axes of identity. However, the theory of intersectionality does more than address an intersectional population, it emphasizes how the experiences of those populations cannot be cleanly reduced to combinations of single-axis experiences and how their right to be treated as full members of each axis are restricted. On these points, InclusiveMag fails, because it specifically seeks to reduce intersectional identities to similar facets as single-identity ones.

Othering

When discussing people, word choice is important. GenderMag describes failures of inclusivity as bugs. The word “bug” carries a

specific connotation: a flaw that was unintended, unanticipated, and often isolated, able to be fixed easily without extensive modification. The use of the word “bug” invites the audience to think of software inclusivity in similar terms, in contrast to narratives about structural issues or intentional exclusion. At the same time, the word “bug” implies something which, though it must be fixed, is of secondary importance to some other goal, which can serve to devalue and other the person who is excluded.

Comparison: QuickCheck

GenderMag has a strong parallel in the world of programming languages: type-directed test case generation libraries. The most famous of these is the Haskell library QuickCheck, which has later been ported to a variety of typed functional languages. The QuickCheck library lets the programmer provide a correctness property and then follows the types to generate test cases, and tests whether the property holds on the test cases.

For example, if a user wants to test that integer addition commutes:

```
test(x : Int ,y : Int): Boolean = (x + y == y + x)
```

then a QuickCheck library would generate pairs of numbers (x,y). QuickCheck knows that it can't possibly hope to test all numbers, but also knows that it would be insufficient to let a single value stand in for all. Not unlike the use of personas, it has a “generator” for each type which tries to generate a diverse, but finite set of test values. For example, one set of test values for integers might be: {-12345, -64, -5, -2, -1, 0, 1, 2, 5, 64, 12345}. This set covers some of the most common “edge case” values and also some larger ones.

Then, for tests with pairs of values, we draw pairs of values from the test set. We gave 11 integer test values, resulting in $11^2=121$ possible pairs, including both edge cases like (0,0) and general cases like (-5,64).

QuickCheck navigates type abstraction in a non-trivial way. Because it runs concrete tests, it *cannot* test types in the abstract. It must test code on some specific integer, not “an integer” in general. However, type structure actually constrains which tests are needed. Parametric polymorphism, in particular, guarantees that program behavior is agnostic to certain changes in typing, a phenomenon that Philip Wadler popularized as “Theorems for Free”.

For example, consider the identity function id, which has type $t \rightarrow t$ for all types t. Because the same function has type $t \rightarrow t$

`t` “for all t”, its behavior cannot possibly depend on t, and so for testing, it suffices to test a simple type such as `t = unit`. Likewise, if a function works for all list types `List[t]`, it suffices to test simple choices of t, drastically reducing the complexity of testing.

Related Work

The potential of connecting programming languages with feminist theory has been explored by feminist scholars since at least 2013, when Ari Schlesinger led discussion of the topic on a blog post published through HASTAC, a major community of humanities scholars. Follow-up work identified feminist themes in the following projects:

- * Mezangelle, an anti-programming language design intended not for writing executable code, but for poetic and artistic writing
- * The *femme Disturbance library*, an unpublished code-like text by the transfeminist scholar micha cárdenas
- * Jailbreaking the Patriarchy, a Chrome extension which alters gendered language in web browsers

The above works prioritize the values of the Humanist: their goal is to provide creative artifacts which can serve as a point of discussion and reflection for the relationships between gender

and technology. Orthogonal to that goal, this chapter focuses on potential needs of the Practitioner and how to meet them.

Discussion of related work would be incomplete without observing that there are limited publications about feminist perspectives on programming language design in part because of individual and collective harassment of scholars by anti-feminists. As of this writing, searching for feminist programming languages does not return results about these prior works nor the closely-related topic of gender in HCI. Instead, the most prominent result is an anti-programming language “C Plus Equality” developed by anti-feminist trolls, and feminist scholars of programming languages often face harassment on the individual level. This harassment obstructs the possibility of free and open interdisciplinary research between the fields of feminist theory and programming languages.

Gender is thoroughly studied in the context of HCI, resulting in thorough guidelines for the treatment of gender in software interfaces.

Additional citations:

Can feature design reduce the gender gap in end-user software development environments? 2008 Grigoreanu “Intersectionality Goes Analytical: Taming Combinatorial Explosion Through Type

Abstraction“ by Margaret Burnett, et. al.

<https://www.femtechnet.org/2014/07/a-feminist-programming-language/> <https://medium.com/@stestagg/how-feminist-programming-has-already-happened-9e4fb507ddb9>
<http://wg18.criticalcodestudies.com/index.php?p=/discussion/11/week-1-gender-and-programming-culture-main-thread>

Classroom Activities

Instructors may wish to tailor their choice of classroom activities to the culture of their classroom, which sometimes varies by discipline. Students in gender studies and related fields are likely experienced in open classroom discussion of gender, while computer science students are often not, and may even be uncomfortable with such discussions. The following list includes both discussion-based activities and design-based activities, for instructors to choose at their discretion:

- * Lead the students in a cognitive walkthrough of a proposed design using GenderMag personas. If the students have prior exposure to cognitive walkthroughs, assign them a walkthrough to perform individually or in groups
- * Present a short text about a topic which is tentatively unrelated to gender. Collaboratively write type definitions

which represent the core concepts that appear in that text. Next, critique and deconstruct that definition, to explore what assumptions it carries or which data it might fail to model

- * Have students use QuickCheck to generate test cases for a program that includes gender data
- * Discuss students' personal experiences of gender in computer science, including discussion of their priorities. When is gender-inclusive software design a priority for gender-marginalized students, and when is it not?
- * Prompt students to share their favorite media containing feminist and/or post-structuralist themes, and explore how those themes might translate to programming languages

Exercises

1. Define a datatype in the programming language of your choice that represents genders
2. Project: Write an essay exploring any one of the topics proposed in the section “Combining Type and Gender”
3. Project: Write an essay exploring the notion of leaky abstractions (imperfect implementations of abstractions which reveal implementation details) in software development and its relation to gender
4. Research project: The study of gender can be compared to the study of genre: genres of media are socially constructed and

evolve incrementally as new media are released. When new media are released, there is typically a consensus as to what genre they belong to, but by being released, they subtly change the meaning of that genre. Design a type that behaves in a similar way to genre.

5. Research project: Use an extensible type, such as the `exn` type in Standard ML, to model the social construction of gender. Your response should include a Standard ML program that compiles and runs.
6. Research project: In the philosophy of language, performative statements are statements which do not have a traditional truth value (they are not true or false), but rather make themselves true through the act of speaking them. Programming languages have related concepts such as statements, Booleans, and mutation, but they do not have a direct counterpart to the notion of a performative statement. Design a programming language that has one.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter19

Chapter 14

Disability

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

Inclusivity for disabled people is a core component of inclusive design for any kind of software, programming languages included. Disability issues require particularly careful analysis because they are at the same time universal and specific: most people will be disabled at some point in their life, but disabled people are a minority at any given point in time. Moreover, because disabilities come in many different forms, individual disabled subgroups such as Blind or Deaf people are minorities throughout their lives.

This chapter draws on and integrates multiple bodies of literature on disability. We draw on literature from disability studies, a field of critical studies which studies, for example, the systemic exclusion of disabled people (ableism) and the political and ideological perspectives of disabled activists. We also draw on literature from disability HCI, which has only recently incorporated perspectives from disability studies, and, as it developed accessible technology, has often treated disabled users as a problem to be fixed from the outside. The goal of this chapter is to thread the line between these studies, by providing actionable programming language design recommendations, but doing so from a perspective that centers disabled experiences.

Disability Studies for Programming Languages

This section highlights critical perspectives which can serve to inform the design of programming languages. One of those perspectives (programming languages as infrastructure) is a new concept first introduced in this book.

Programming Language as Infrastructure

Programming languages are the infrastructure of computing. In contrast to application software, a programming language is not

an end in its own right. Instead, a programming language is a means to many different ends. A comprehensive understanding of programming language accessibility, as with infrastructure accessibility, includes both an analysis of equitable access to destinations and the experience of a disabled person on the way to their destination. We raise transportation infrastructure as an example:

- * Equal access to destinations means that disabled people can reach the same destinations as abled people. An example of equitable transportation access is that if an abled person who drives and disabled person who cannot drive work in the same building, there should be another transportation method that can take the disabled worker to the same building. An example of equal access in programming language design is that if a Blind programmer relies on screen readers to read code, they should be able to extract the same information about the program as a sighted programmer.
- * Equitable access to destinations recognizes that disabled people may travel to different destinations from abled people with different frequency. Many disabled people must visit medical providers frequently, so those destinations may be priorities for accessible public transit. Programming languages can recognize that disabled programmers sometimes have different programming goals from abled programmers. As discussed in the Crip Technoscience section,

disabled programmers have an outsized need to write programs which improve access.

- * Equity of the travel experience recognizes that infrastructure should allow disabled people the same comfort and safety during their journey as abled people. Transportation infrastructure can fail by providing access without comfort, e.g., by allowing pedestrians to get splashed with water by drivers, by providing uneven sidewalks that are difficult to navigate on wheelchairs, or by allowing harassment on buses or trains. Likewise, programming languages fail as infrastructure if they expose disabled programmers to constant inconvenience or potential harassment from peers.

The metaphor of programming language as infrastructure not only highlights the potential ways disabled programmers can be excluded, but highlights the direct material impacts of accessibility. Infrastructure is opportunity. History has proven that cities with major transport connections such as trains or highways will grow, attracting stable careers, population, and cultural growth. It is common knowledge in computing culture that computing careers are viewed as such opportunities, thus accessible programming languages are the road to that economic opportunity. As with infrastructure, there are additional destinations beyond economic ones: programming languages as infrastructure can enable full participation of disabled people in

other tasks that use computers, including non-computing careers and social activities.

Visibility

The tension between visibility and invisibility is a core concept in disability studies. Invisibility and visibility each carry their own risks. When a person's disability is invisible to the people around them, they are less likely to receive accommodations that support them in completing activities of daily living and may experience increased social isolation, which is a key predictor of long-term health outcomes. While visibility can improve access to accommodation and social companionship, it can also increase exposure to social discrimination, including housing, employment, and education discrimination as well as harassment.

Because of these competing benefits and dangers of visibility vs. invisibility, it is commonly advocated to provide disabled people agency over whether they are visible or invisible. At the same time, it is recognized that the difficulty of achieving agency depends on the specific disability involved and who is doing the viewing.

Disability Spectrum

Disability studies sees disability as a spectrum or a continuum. This does not mean that disability lies on a one-dimension line from “most disabled” to “most abled.” Instead, this spectrum is multi-dimensional. Different dimensions of disability can include sensory, physical, neurodevelopmental, and intellectual disabilities, all of different severities. These dimensions interact with other dimensions of a person, including their level of social privilege or disprivilege. A spectrum is also something that a person can move along. Some people have dynamic disabilities, where their abilities are impaired greatly on some days and minimally on others.

These complexities lead to a strong emphasis on avoiding generalizations about the disabled experience and avoiding neat divisions of disabled people into rigid categories. Advocates of integrating disabled students with abled students have pointed to this fluidity as reason for integration. At the same time, integration is not the same as assimilation, and many disabled people who view their disability on a continuum still find community in distinct disabled sub-cultures separate from mainstream abled culture.

Crip Technoscience

Crip technoscience applies the “crip” perspective to technoscience. The crip perspective is an anti-assimilationist one, which views disability as a desirable element of the world which should resist being “fixed”. Technoscience views science, technology, and politics as connected, where the creation of science and technology can be a form of political action.

In particular, crip technoscience emphasizes that disabled people are makers and inventors, who routinely create technology, modify it, or use it in novel ways to meet their access needs. In doing this on their own terms, they promote equity for disabled people.

Perspectives from the Global South

Intersectional disability studies have explored the lived experiences of disabled people in Global South countries, including Kenya, Uganda, and Jordan, and explored how the priorities of disabled people in these countries can differ from those in Global North countries. These studies have identified that lack of legal protections and the extent of disability stigma in participants’ daily lives produce a qualitatively different experience from Global North countries such as the United States.

For this reason, the social aspects of disability took on a pronounced significance compared to the functional aspects. Visually-impaired Kenyans highlighted the need for blending functional improvement with community education about disabled people's needs. For example, in a speculative design exercise, the best-received idea was an intelligent traffic light that alerted able drivers to disabled pedestrians' need for extra time to cross the street safely. In a separate case study by the same researchers, Kenyan wheelchair users pointed to wheelchair aesthetics as key to improving their self-esteem. Jordanian and Uganda prosthetic wearers presented the flip-side to community education: because ableism is prominent in their lives, they expressed the need to blend in for safety. Aesthetics of prosthetics, such as matching the skin color and size of the wearer, *are* a practical concern, because they allow wearers to blend into able society for their own safety. The study frames these needs as a need for individuality over functionality.

It is important to recognize that the notion of individuality is itself context-sensitive. In the United States, individuality often corresponds to visibility, self-determination, and the freedom to ignore what the people around us think. In contrast, the study subjects called for a context-sensitive approach to accessibility which allows them to prioritize local, community-specific needs of the individually. Likewise, the role of individually in design

depends on the thing being designed. In the context of programming language design, the need to blend in could manifest through the design of syntax or accessible editing software, for example.

Survey of Disability Issues in Programming Languages

This section explores accessibility issues faced by different categories of disabled people when using programming languages.

Musculoskeletal

Repetitive strain injuries (RSIs) such as Carpal Tunnel Syndrome and tendinitis are the largest occupational health risk for professional programmers. Depending whether RSIs are acute vs. chronic, they can be disabilities. Moreover, the accessibility issues around RSIs overlap significantly with those for life-long connective tissue disorders such as the Ehlers-Danlos Syndromes (EDS).

If programming languages are viewed as interfaces, then musculoskeletal accessibility is a problem of making the input mechanism accessible. This interface could be made accessible either by reimagining the physical input mechanism (ergonomic

keyboards, speech-to-text, mechanical inputs for muscular dystrophy patients) or by changing the design of programming language syntax.

Check your understanding: Ergonomically, how does programming language syntax differ from other text?

Answer: Programming language syntaxes usually make heavy use of punctuation. On a standard QWERTY keyboard layout, many of the punctuation keys are located at extreme positions.

One of the basic principles of ergonomics is to reduce extreme motions, and standard programming syntaxes on standard keyboard layouts can produce such motion. This concern is not merely an abstract one: custom keyboard layouts such as Programmer Dvorak have been developed to reduce extreme motions for common programming language syntaxes.

Keyboard changes cannot address all RSIs, let alone all musculoskeletal conditions. Speaking programmers with RSIs may require speech-to-text support, while non-speaking programmers may require specialized mechanical inputs, as have been used by programmers with muscular dystrophy.

Visual

If musculoskeletal disabilities interact with input devices such as keyboards, then visual disabilities interact with output devices used to read code during editing and read outputs during execution. The preferred choice of output device varies by the extent of vision: fully blind people typically need screen readers to read text aloud, but the majority of visually disabled people retain some vision and may rely on accessibility features such as screen magnifiers, large font sizes, or even high-contrast color schemes instead of or in addition to screen readers. We first discuss accessibility issues facing screen readers, then broader issues affecting visually disabled programmers.

Whitespace Sensitivity

Traditional screen readers work by reading aloud the non-blank characters in a text and ignoring whitespace. Because they do not read spaces, they will fail to communicate the full content of code in whitespace-sensitive programming languages. Whitespace can determine the meaning of programs through indentation, notably in Python (and in Haskell), or through assigning special meanings to specific columns in older programming languages such as COBOL or Fortran. As an example, consider the following Python

program, where B1 and B2 are boolean conditions, and S1, S2, and S3 are statements:

```
if B1:
```

```
S1
```

```
if B2:
```

```
S2
```

```
else:
```

```
S3
```

In the above program, S2 and S3 belong to an if-else block nested within the if B1 block. By removing indentation from the `else:` block, it can be made into the else branch of `if B1`, fundamentally changing the meaning of the program while leaving the pronunciation unaltered.

```
if B1:
```

```
S1
```

```
if B2:
```

```
    S2
```

```
else:
```

```
    S3
```

This ambiguity makes it impossible for the user of a traditional screen-reader to determine the meaning of a program. Python's response to this limitation is that it provides "begin" and "end" keywords which can be used to write whitespace- insensitive code. However, this approach fails on two points: it does not assist screen-reader users in reading other people's code, and it does not respect their right to control their own visibility, because it encourages them to program using a syntax that is visibly different from other programmers.

Ambiguity Across Operators

We now explore how ambiguities in the pronunciation of syntax occur even for whitespace-independent languages.

Check your understanding: Suppose I am reading a program aloud to you and I say "two". What program am I talking about?

Answer: I could be referring to the number literal 2, the variable two, or the string “two”, so the answer is ambiguous. Homophones make this even more ambiguous, introducing the possibilities “to”, “too”, to and too.

To allow full participation of visually disabled programmers both as readers and as writers, specialized screen readers have been developed which disambiguate the pronunciation of each expression, e.g., pronouncing “number two,” “string two”, or “variable two”. These customized screen readers allow programmers to read existing code and write code in pre-existing syntax.

Textual vs. Visual vs. Tactile Syntax

Though the most popular languages for professional programs are textual, visual syntaxes for programming languages, such as flowchart-based notations, have received significant research attention. These syntaxes have been used to:

- * teach programming to children (notably through the language Scratch),
- * to enable domain-specific programming by professionals who are not programmers (such as scripting interactive sequences in video games with the Kismet language, or visual notations for effects in video-editing software), and

- * to serve as visual communication aids for a textual program, as has often been done with the modeling language UML.

The design of visual programming languages aims to improve access of new groups to programming languages, but its reliance on vision poses the potential to exclude visually-disabled programmers. Before dismissing visual programming languages out-of-hand, insights from disability studies encourage a closer look. A full understanding of accessible programming language design requires understanding disability as a spectrum, where fully-blind, partially-blind, and fully-sighted programmers might all work together, using the same set of tools. This continuum of disability has been supported by several styles of syntax, including tactile syntax and hybrid visual syntax.

Tactile Syntax: Torino

Tactile (touch-based) syntax has been explored through the programming language Torino. The design goal of Torino is to teach programming and computational thinking to children ages 7-11 across the spectrum of visual disability.

Torino's design translates the concept of visual flowchart-based or block-based design into a tactile paradigm, where the components of a program are represented as physical beads that can be connected to one another by visual wires. A given bead may be

responsible for input via a knob, output via sound, conditional if-else branching, or iteration. By combining the beads together, students can produce basic programs.

A successful tactile syntax must enable the programmer to distinguish the beads for each operation. This is achieved by giving a distinct physical shape to each bead, which the students could consistently tell apart. In addition to this design choice, the use of physical wires enabled students to browse through a program they had already written, physically scanning a chain of connected beads to identify the bead they wish to modify.

The developers of Torino emphasize its potential use in settings with a spectrum of disability, and evaluated it in that context. The evaluation showed that students across this spectrum interacted with the syntax in different ways: students with greater vision would rely on their vision instead of touch to identify distinct bead types. However, students with different vision levels could collaborate effectively in a shared classroom setting.

Including programmers across the full spectrum of disability has important social and emotional consequences. For disabled people of all ages, but children especially, social isolation resulting from ableism can have significant impacts on life satisfaction and long-term health. Enabling a broader range of programmers to program collectively in their own preferred style does not

minimize or erase fully-blind programmers, rather it provides an opportunity for building social connections.

The importance of building social connection through the structure of a language is not original to Torino. This social value traces back, for example, to the development of protactile sign language, a touch-based sign language originating in the communication of DeafBlind women in Seattle. Protactile sign language centers DeafBlind people by providing richer tactile communication than traditional sign languages, while still being learnable by sighted people. Likewise, Torino gives priority to rich tactile interfaces while still supporting the use of vision. This is not to say to say Torino and protactile should be considered directly comparable: one is a technology, the other a social practice.

Hybrid Visual Syntax

Recognizing that text is the dominant medium for writing code, but that images have unique strengths in representing many computational concepts, hybrid visual syntaxes have been developed, where the same program simultaneously has equivalent textual and visual semantics. Potential applications include trees, matrices, filesystems, network messages, videos, and circuits. By endorsing both a textual and visual semantics at the same time, these syntaxes enable domain-specific interactive

visual editing while seamlessly integrating with text-focused tools such as textual editors and source control. Though the authors of this work do not explicitly cite collaboration between visual programmers and textual programmers as a motivation of the work, the work can be used to that end.

Related Work

<https://www.tandfonline.com/doi/pdf/10.1080/07370024.2018.1512413> http://katta.mere.st/wp-content/uploads/2020/03/Individuality-over-function_revised.pdf

Nothing About Us Without Us

Disability: The Basics, Thomas Shakespeare

disability reader

Crip technoscience manifesto

<https://catalystjournal.org/index.php/catalyst/article/view/29607/24771>

<https://sites.middlebury.edu/unquietminds/files/2013/02/defining-mental-disability.pdf>

Adding Interactive Visual Syntax to Textual Code

Language Note

Members of the disability community differ in whether they prefer to be described with identity-first language (disabled person) or person-first language (person with a disability). This chapter uses identity-first language because it is the preference of the disabled author. When talking to or about a person who prefers different language, their preference should be respected.

Acknowledgement

Special thanks to Leif Andersen for sharing her experiences as a legally blind programming languages researcher with the author in conversation. The ambiguity example “two” is credited to Leif.

Classroom Activities

- * Use a screen reader to read out the ambiguous sample programs from this chapter as a demonstration
- * Look at a keyboard with students and explore which commonly-used programming syntax relies on difficult-to-reach keys

- * Live-code a short program using a hybrid visual syntax
- * If the class contains a substantial population of disabled students who are comfortable openly discussing their experiences of disability in computing, invite them to share. Do not force students to share, especially if they are a small minority of the class

Exercises

1. Consider the toy programming language from the Types chapter. Write a function which accepts an abstract syntax tree in that language and produces a string of English words which is different for every distinct syntax tree
2. Prove by induction that your function produces distinct output strings for distinct inputs
3. Write a program which takes in a file and counts the frequency of every keyboard character in the file. Print the frequency counts in a grid, in the same order as the keys of your keyboard
4. Call the program from the previous exercise with a program you wrote as the input. Assess the relative frequency of characters in your program against the difficulty of typing each character. Which hard-to-reach keys appear frequently?
5. Design a new programming language grammar reduces the use of hard-to-reach keys for the program from the previous

exercise

6. Convert your program to use the new syntax from the previous exercise. Compare it to the original syntax. Count the number of characters and lines in each. Is one significantly more concise than the other? Is one significantly more readable than the other?
7. Project: Design a physical artifact which represents a particular program or algorithm through touch

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter20

Chapter 15

Media Programming

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * We explore the design values underlying the programming language Processing
- * We explore the role of programming environment design in programming language design
- * We discuss the notion of continuity across languages, the ability to transfer expertise acquired for one programming language into another

Media artists, such as visual artists, can use programming to assist them in the creation of art, whether static, animated, or even interactive in nature. As a programming

language designer, how would you design a programming language and its environment to meet the needs of media artists?

We explore this question through a case study on Processing, a prominent programming environment and language for media arts. Because Processing targets a specific community of programmers, it can be considered a domain-specific language. More than anything, however, our discussion of Processing will blur the lines between development of new languages, new libraries, and new programming environments. Processing is notable less for its differences from previous programming languages, and more notable for how it shares a deep relationship with them, while succeeded in reaching a community of programmers that prior languages did not. That success points to the fact that there is more to a programming language and its community than just syntax and semantics.

Example Program: “Saturation”

We present an example program in Processing, from the standard set of examples. Processing provides a web-based interface for editing and executing programs; you are encouraged to [run](#) the program in your browser before reading the explanation of the code below.

```
int barWidth = 20;
int lastBar = -1;

void setup() {
    size(640, 360);
    colorMode(HSB, width, height, 100);
    noStroke();
}

void draw() {
    int whichBar = mouseX / barWidth;
    if (whichBar != lastBar) {
        int barX = whichBar * barWidth;
        fill(barX, mouseY, 66);
        rect(barX, 0, barWidth, height);
        lastBar = whichBar;
    }
}
```

Example program published by the Processing developers, which plays with color saturation

This program is an exploration of color. In an HSB (aka HSV) color space, each color is represented by its hue, saturation, and brightness (aka value). Saturation is a number that is 0 when the color is completely gray and is at its maximum value when the color is as not-gray as possible. The program displays a rainbow pattern and interactively updates the saturation of each column whenever you move the mouse.

Processing uses a C-like syntax. Variables can be defined outside of functions, and must be given types. This program defines integer variables `barWidth` and `lastBar`, respectively for the width of bars to draw and the x-position of the most recent bar drawn.

The most important functions in a Processing program are named `setup()` and `draw()`; they have no arguments or return values. The `setup()` function is called once at the beginning of the program to set up the drawing area, etc. The `draw()` function is run repeatedly to draw the screen, and is the main part of the program.

Here, the `setup()` function sets the canvas size, sets an HSB color space scaled to a maximum value of 100 and `noStroke()` indicates not to draw the outlines of shapes.

The main `draw()` function checks whether the mouse has moved. If so, it sets the fill color based on the Y coordinate of the mouse, draws a rectangle in that color, and remembers the mouse position.

Check your understanding: To test your understanding of this Processing program, make a small change in the online code editor, such as changing the brightness to 75%. Rerun the edited program in the online code editor and observe the changes.

This completes discussion of the example Saturation program, which is a complete, executable Processing program. This example suffices to demonstrate the basic nature of programming in Processing.

Design Values

If you know C and you read a Processing program, you might wonder what new features Processing contributes when compared to C. From the perspective of syntax, and to a less extent, semantics, there would indeed be little new. However, it is also a historical fact that Processing achieved something C did not: it brought many new creatives into the fold of the programming world. There must be a reason that it succeeded in this goal. To explain the success of Processing, we should look at its design objectives, which differ greatly from those of C and make the *holistic experience* of programming in Processing wildly different from the experience of C programming. The design objectives listed here are a mix of those explicitly stated by the Processing developers and interpretation by the author.

Visuality

Processing is unapologetically focused on media, especially visual media, to the point that the “main” function of a program is a

“draw” function. Common programs involve visual effects applied to external images, particle systems, and fractal systems.

As discussed in the Disability chapter, exclusive focus on visuality can hinder accessibility for visually-disabled programmers, though because Processing syntax is text-based, its greatest accessibility is accessibility of visual output, which is a more fundamental challenge to overcome than accessible text.

To motivate the focus on visuality in Processing, we discuss the concept of design continuity.

Continuity with Target Audience

The design of a domain-specific language should always be centered on the background and culture of its target audience. By centering visual media, Processing centers something its intended users already know. Drawing is treated as a foundational art form in traditional art education; even artists who work in other media typically know drawing. We name this principle *continuity*: the design of Processing is made continuous with something the audience already knows, in order to reduce the amount of new learning required when learning to program in Processing. The word “continuity” highlights that programming is by no means the same activity as drawing: programming requires a distinct skill

set, but Processing merely connects it to artistic concepts when possible.

Even aside from Processing, visuality (often in input rather than output) has been a common focus for programming languages which aim to be approachable for novices, with entire research conferences such as VL/HCC dedicated to the topic. This trend can be motivated through continuity, as vast numbers of people have experience, though not expertise, with drawing, but may also be motivated from other perspectives:

- * Visual programming is used to teach programming concepts to younger children who have not yet developed adult-level literacy
- * Many data types have well-established visual representations, which may be more convenient for editing and display than textual representations
- * Visual editors can be designed which give faster feedback on changes to a program, benefiting productivity and motivation

Remixable Openness

An academic paper introducing Processing talks about the open source community, and how open-source development had

previously made major impacts in other areas of software, but not media software.

We coin the phrase “Remixable Openness” to point out that Processing is not merely a programming language with an open-source implementation; it aims to do something more. Artistic communities have had their own version of open source long before programmers did, because imitation is a fundamental part of art. The style of code-sharing in Processing is closer in practice to the use of StackOverflow to copy-paste code snippets, except (1) it predates StackOverflow by several years and (2) reuse is viewed positively, whereas the use of StackOverflow is sometimes viewed as a source of shame for other programmers. In short, open source for creative coding looks like copying others code, experimenting with it, and making it your own.

This requires community effort and technical infrastructure (the Processing website makes copy-pasting easy) but it also places implicit requirements on the programming language design. If you have ever copy-pasted code from StackOverflow, you know that significant changes are often required to integrate it into your own program, because the languages and libraries involved are so complex and interdependent. For sharing to work, the language and its standard library must be, in some vague sense, simple.

Immediacy

An essential but rarely-named aspect of the programming experience is immediacy. Immediacy means that a programmer can start programming in a very short period of time and see their first program outputs in a very short period of time. Immediacy is known to be important to student morale in educational settings, so it is a common feature among educational programming languages. The Racket programming language is another example of a language used in educational settings that provides immediate output to a new user: program expressions like

(`+ 1
2`) can be quickly evaluated to their outputs (in this case, 3).

Processing achieves immediacy through a web-based editor (before they were popular), and through a highly visible library of short examples.

Continuity with Other Languages

The C-like syntax is intentional, and it is a feature, not a bug, again reflecting the design value of continuity. What this means is that when designing something new, it should be entirely new, but instead its overlaps with existing work should be just as clear as its novelty.

In mathematics, we can think of a function as being continuous from the left(-) or continuous from the right(+). In design, likewise one can consider two kinds of continuity:

Continuity with “before”: Many readers probably knew C *before* they learned Processing. You experienced continuity with “before”. Because they knew C, it was probably easier to learn Processing. This kind of continuity is important when learning most new things. Even if a language contains many features, our goal is to add only a modest number of concepts to a person’s mental map of programming. If we add too many new concepts (for example, by duplicating existing computing concepts using new terminology of our own) we increase the amount of learning required without providing new functionality. Many languages designed today target this style of continuity.

Continuity with “after”: Processing actually targets a different kind of continuity. Recall that its intended audience are people who have never programmed before, so C-like syntax does not help them learn - there is no biological mechanism which makes humans learn C syntax faster than any other syntax.

The designer’s point is a far more ambitious one. Just because someone has never programmed *before* does not mean they will *never program again!* The designers recognize that there is not one future career path for their users, but multiple. In their

future art careers, some of them may only ever need Processing. But others may find a day where they need general-purpose programming for an advanced project, and others may decide to become full-time software developers. For these latter categories, C-like syntax was important because it means that once someone has learned Processing as their first language, they can transfer their knowledge to other languages like C. The gap between the two languages is real: concepts like pointers are no easier to learn, but prior experience will allow these programmers to focus solely on these differences when learning C.

This is a marked improvement over earlier programming languages used in education. Since the early days of programming, we have known that it is a difficult skill to learn and sought to make it easier. The first educational language is generally considered to have been released in 1964: the original BASIC.

Consider “Hello World” programs in four languages used for teaching. (Source for all examples: RosettaCode.org)

```
BASIC(initial release: 1964; example is BASIC256 dialect)
clg          # Clear the graphics screen
font "Arial",10,100 # Set the font style, size, and weight r
color black      # Set the color...
text 0,0,"HelloWorld!"  # Display in (x,y) the text HelloWor
```

```
LOGO(initial release: 1967)
print [Hello world!]
```

```
Scheme(initial release: 1975; example is R7RS dialect)
(import (scheme base)
        (scheme write))
(display "Hello world!")
(newline)
```

```
Racket(initial release: 1995)
#lang racket
(displayln "Hello world!")
```

“Hello world” example program in four programming languages used in teaching

All of these are markedly different from C syntax. If you already know many programming languages, you probably read all of ~~these without problem, so what's the big deal? The ease of reading~~ is actually a side effect of expertise. Once you have an extensive and well-connected mental model of programming, it is easy to rapidly integrate new information such as differences in syntax. For a student of one programming language, however, this knowledge map may be limited, and transferring information from your first language to second is much harder than transferring it from ninth to tenth. These languages cannot necessarily be faulted. Lisp was once popular, and is the ancestor of Scheme, Racket, and Logo. Basic had no good role models in its early days, and instead became one of the world’s most popular languages in its own right.

Contingency in design

Many texts on programming languages omit comparisons of syntax differences between programming languages. They do this intentionally and proudly, because those differences are insubstantial from the Theorist's perspective. Rather than calling these differences insubstantial, we apply a more precise term: syntax choices are *contingent*. Contingency means that no natural law forced programming languages to use the syntax they do, rather syntax arises from human choices which could have been resolved differently. Though contingent choices can arise in the design of type systems and semantics as well, the degree of contingency is reduced because type systems and semantics are constrained by competing design concerns such as type safety. Designers should be aware of the distinction between contingent design choices and non-contingent design choices. When resolving a contingent design choice, a designer might consult competing historical design choices and make their choice based on a desire for continuity.

Related Work

Today's reading is: Processing: programming for the media arts (2006). <https://link.springer.com/content/pdf/10.1007/s00146-006->

0050-9.pdf?pdf=button The main website for Processing is
<https://processing.org/>

rosetta code

processing saturation example

Classroom Activities

- * Demonstrate the Saturation example program in class
- * Allocate time for students to tinker with another Processing program

Exercises

1. As you program in Processing, take notes about your usage of the interactive Web editor. Do you write large amounts of code before testing, or do you test frequently after small changes? Record any benefits to your approach that you notice, such as bugs caught
2. Projects: Implement specific programs in Processing:
 - a. A picture of a unicorn made from basic shapes
 - b. A Sierpinski triangle
 - c. A Game of Life simulation

- d. A Mandelbrot set viewer
 - e. An interactive synthesizer where the user can select sine, triangle, square, and sawtooth waves from which a sound is generated and played
3. Assemble a mood board of your favorite art pieces made with Processing
4. Develop an original art piece in Processing inspired by the mood board you created in the previous exercise. If those art pieces are released under licenses that allow derivative works, consider incorporating their code into your art piece, with citation

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

[bookish.press/book/chapter23](#)

Chapter 16

Play

by Rose Bohrer × + author

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

How do programming languages show up in the world of play, in digital media such as video games and interactive fiction? Certainly, many digital media *are* programs, and programming languages are used to implement them, as well as to implement the engines that are used to build games.

In the first half of this chapter, we take a playful approach to exploring the relationship between programming languages and digital media, particularly interactive fiction. As a case study, we will use programming language syntax and semantics to model Twine, a code-free tool for developing interactive fiction. In doing

so, we illustrate how broad the notion of a programming language can be, while also exploring how theoretical results about programming languages can be developed even for seemingly-exotic applications.

The second half of the chapter is more focused on the social context of interactive fiction, and how it can be used not only as entertainment, but how it can be used for communicating personal narratives in important contexts such as academic research.

Twine is an open-source, web-based environment for developing interactive fiction (such as text adventures and visual novels), first released in 2009. One of its selling points is that it allows new users to create games without programming, strictly using the graphical interface, but then to add more complicated functionality using HTML, Javascript, and CSS if and when they're ready to. We will explore the core language, how to model its extensibility, and how Twine has been used.

Check your understanding: Play a short Twine game. What mathematical structure best explains the structure of a Twine game?

Answer: The basic structure of a Twine game is a directed graph.

Formal Semantics of Twine

We develop a formal operational semantics for Twine based around the understanding of Twine games as directed graphs, so that we explore theoretical properties of Twine games. To interpret a Twine game as a directed graph, we treat each passage of text as a node and each directed edge represents a link in the text from one passage to a successor passage. For example, if a node has two out-edges, then the player gets to pick which passage of the two passages of text comes next, and if the node has one out-edge, the player gets no choice. In serious Twine usage, it is essential that the language is extensible beyond basic graph structure, using HTML, Javascript, and CSS. However, that adds complexity to the formalism, so we begin with a non-extensible Twine and then add extensibility.

Non-Extensible Semantics

The runtime state of a Twine program consists of a program graph G and current state s . For use the variable m to range over these runtime states, short for “machine configuration”:

$$m \leftarrow (G, s)$$

where a graph G consists of vertices and edges

$$G \leftarrow (V, E)$$

and the start state identifies a vertex. Each vertex $v \in V$ is a pair (n, p) of vertex identifier n and passage of text p . We write $VN = \{n \mid \text{exists } p \text{ such that } (n, p) \in V\}$ for the set of vertex names, then the start state satisfies $s \in VN$.

Each edge $e \in E$ is a triple (u, v, l) where $u \in VN$ is the source, $v \in VN$ is the destination, and $l \in String$ is a label identifying the edge. A state s is just a string $s \in VN$. We do not develop a full type system for Twine, but we make basic well-formedness assumption about programs, such as $u, v \in VN$ for all edges in E .

The operational semantics of Twine will seek to explain which vertices are explored as the player plays the game, step-by-step. Our first observation is that because we execute the program step-by-step, we will need a small-step semantics as opposed to a big-step semantics. Secondly, we observe that the vertices explored depend on the player's actions. Actions are indicated by edge labels l , and an entire playthrough of a Twine game with length k is indicated by a sequence l_1, \dots, l_k of k edge labels. We write ls for such a sequence of labels.

We write $m \mapsto_l m'$ to mean that m steps to m' in one step along label l and $m \ done_{ls}$ to mean that m is done executing, where ls is the sequence of remaining actions to be executed. We write m

\mapsto^*_{ls} , where ls is a sequence of k labels, to mean that m steps to m' in k steps, each following the given edge label. Then, the semantics of Twine programs are defined by the following rules:

The judgement $m \ done_{ls}$ consists of a single rule: execution is done if no actions remain to be executed.

Rule DoneEmp

*

$m \ done_{ls}$

(where ls is the empty sequence)

The judgement $m \mapsto_l m'$ has a single rule:

Rule StepOne

$G = (V, E)$

$(s, s', l) \in E$

$(G, s) \mapsto_l (G, s')$

and the judgement \mapsto^* is defined in terms of the prior two judgements:

StepsDone

$m \ done_{ls}$

$m \mapsto^*_{ls} m$

¶

Rule StepsNext

$m \mapsto_l m'$

$m' \mapsto^*_{ls} m''$

$m \mapsto^*_{l\ ls} m$

In rule StepsNext, the notation $l\ ls$ means a sequence whose first element is l and whose remaining elements are collectively named ls .

Automata Theory

This computational model of Twine is surprisingly similar to a specific formal model of computation, which comes not from programming language theory but from automata theory (If you do not have prior exposure to automata theory, consulting standard texts may assist in reading this section)

Check your understanding: What formal model of computation are Twine programs closely related to?

Theorem: Not-extensible Twine is equivalent to deterministic finite automata (DFAs), once extended in the following fashion:

- * DFAs divide their states into *accepting* and *non-accepting* states. The equivalent of an accepting state in a game is a victory state, i.e., a state in which the player has won the game. Thus we define a set of victory states.
- * In DFAs, every action can be applied from every state, but this is typically not the case in Twine games. Thus, we define a new stuck state and, for every previously-undefined action, define it to go to the stuck state.

Proof: Exercise.

Once we establish the formal relationship between Twine games and DFAs, we can apply classic results from the theory of DFAs to establish surprisingly direct results about the behavior of Twine games.

Corollary: *The number of moves in the shortest winning play is at most the number of prompts in the game.*

Proof: By the Pumping Lemma.

Corollary: *If any winning play contains a cycle, then arbitrarily long winning plays exist.*

Proof: By the Pumping Lemma.

Corollary: *The set of all winning plays forms a regular language, i.e., there exists a regular expression which detects all winning plays.*

Proof: By definition of regular languages, which are the languages of DFAs.

These theoretical results were not designed into the fabric of Twine, and it is likely that most Twine developers never even consider these results while developing their Twine games.

Rather, this overlap between Twine and DFAs is a happy coincidence for the Theorist.

Extensible Semantics

How should we formally define an extensible language? One of the easiest ways to do this is to treat the extensions as entirely abstract. All we know about extended Twine is that there exists some extra state (set ES) beyond the vertex names, and there exist some extended actions that can affect the extended state in arbitrary ways (write $action(s,l)$ for the state resulting from label l on state s).

It now suffices to redefine the state s from a vertex identifier n to a pair (n, es) where $es \in ES$ is extended state, and to redefine action sequences so that each l is an arbitrary string, not necessarily one that has an edge out of the current vertex. We need only add one new stepping rule

Rule StepExt

$$G = (V, E)$$

$$s = (n, es)$$

$$(n, n', l) \notin E \text{ (for all } n' \in VN\text{)}$$

$(G, s) \mapsto_l (G, \text{action}(s, l))$

and slightly modify rule StepOne

Rule StepOne

$G = (V, E)$

$s = (n, es)$

$(n, n', l) \in E$

$s' = (n', es)$

$(G, s) \mapsto_l (G, s')$

Twine in Human Subjects Research

This section discusses a series of papers which employed Twine in human subjects research, specifically research on the design of antiracist technology. The use of Twine was explored in this context as a tool to address the fundamental question: “How should an HCI researcher or practitioner conduct design workshops where the participants will explore their lived experiences of

interpersonal racism, while minimizing re-exposure to racial trauma?”. The answer chosen by the researchers was to explore those lived experiences through making fiction in Twine.

We explore three concepts in HCI that contextualize this work: participatory design (the broader class of research approaches into which the work fits), design fiction, and foundational fiction.

Participatory Design

In HCI research, Participatory Design (PD) refers to many design approaches that seek to democratize the research participants by allowing research participants to play a role in the design of the design artifact. In the least radical forms of PD, participants provide formative input to the design team, who then use it in their design process. In more radical forms of PD, the participants typically play a direct role in design and have direct co-ownership of the design outputs. Even radical PD typically maintains a distinction between designer (or researcher) and the research participants. For approaches that blur that distinction by elevating the participant, see “Research-Practice Partnerships”; for an approach that blurs the distinction by making the researcher the subject, see “Autoethnography,” both described in the [Qualitative Studies](#) chapter.

In the research discussed here, the role of PD is thus: the researchers' goal in running workshops about lived experiences of interpersonal racism is to ultimately design software tools that would be used to support victims of racism in its aftermath.

The participants' participate in the design of those tools, particularly by identifying what the function of that software should be in order to help them. The challenge is that to design technologies in response to any sort of interpersonal trauma, one must first relive and explore that trauma, which poses high emotional cost and risk of retraumatization even in an optimal environment, let alone a setting with strangers. The use of fiction is meant to provide mental distance from trauma and minimize re-exposure to trauma.

Design Fiction

Design Fiction is the use of fiction to explore future possibilities realities, typically to speculate on the role of the design artifact in those realities. Those realities could range from utopias to dystopias, and from social to technical. We could ask:

- * What phone apps would be important in a post-climate-catastrophe world?

- * What online democratic procedures would be best in a totally honest world?
- * How could we use wearables in new ways if we had perfect Internet everywhere on Earth?
- * How should modern software be designed for communities that still have very limited access to modern computing hardware?

In Design Fiction, the role of the fiction in the design is quite direct. It creates a space for design exploration. It performs worldbuilding, then stops, leaving room for the designers to fill in new possibilities.

Foundational Fiction

Foundational fiction stands in contrast to Design Fiction. In contrast to Design Fiction, Foundational Fiction is about participants who exist in the here and now, so named because it provides a new foundation for discussions between the participants. Its key feature is that it provides the participants with greater agency about how much personal information they reveal during the interactions. Fiction always contains an element of its author, and the power of fiction here is that the participants can choose how much of themselves to put into their stories;

whether they write autobiography or fantasy, the other participants might never even know the difference.

The goal of this foundational fiction is to improve sense of safety among participants, but sense of safety is difficult to measure directly. Instead, the researchers measured it indirectly, via another key objective. A major practical end of increasing felt safety was to increase participant engagement and creativity in proposing technological solutions. On these metrics, the authors deemed the work successful: the 26 participants generated 122 ideas, of which they produced storyboards of 20 (as a follow-up step). This metric highlights that Foundational Fiction, like Design Fiction, still plays an important role in generating design ideas, but its role is more indirect.

Why Twine?

The choice of Twine is contingent; the authors could have chosen another tool to complete their study. However, the fact that Twine does not require writing code was likely a major motivating factor.

The goal of the research was to collect knowledge from lived experiences that are widely distributed throughout the populace, not concentrated only in programmers.

Classroom Activities

- * Play a Twine game to warm up the class. If the class is small, play as a group and let students take turns suggesting moves for you to enter. If the class is large, students can play individually or in small groups
- * Draw a small DFA, then translate it to Twine to demonstrate the connection between the two languages
- * Visually demonstrate the Pumping Lemma by drawing cycles in a DFA

Exercises

- * Write a short Twine game
- * Convert the following DFA to Twine: $q_0 \rightarrow a\ q_0 \mid b\ q_1 \mid c\ q_2$; $q_1 \rightarrow a\ q_2 \mid b\ q_2 \mid c\ q_0$; $q_2 \rightarrow a\ q_2 \mid b\ q_2 \mid c\ q_2$. In this notation, semicolons separate the definition of each state, vertical bars separate the rules for each character, and a character followed by a state indicates that the given character leads to the given state
- * State the DFA-Twine correspondence theorem formally
- * Prove the DFA-Twine correspondence theorem

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter24
Chapter 17

Natural Language

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

Imagine telling a machine what you want it to do in plain language and watching it spring to life before your eyes to carry out your commands. This dream has captivated programming language designers from the earliest days of computing up to the present day, the dream of natural-language programming (NLP, not to be confused with natural language processing). Forming a meaningful program syntax from fluid human language is a major design challenge, and making the task of programming as simple as the task of speech has been a perennial pipedream.

We trace NLP from its origins in early computing, to modern NLP languages, then speculate on its potential futures:

- * We trace the origin of NLP to the early language FLOW-MATIC
- * We present Inform, a language for interactive fiction, as an example of modern, successful NLP in a specific application domain
- * We compare and contrast trends in Natural Language Programming vs. Natural Language Processing to explore potential futures for NLP.

Early NLP: FLOW-MATIC

The effort to integrate natural language into a programming language began with one of the earliest high-level programming languages, FLOW-MATIC. For context, FLOW-MATIC was developed in 1955, earlier than the best-known early programming languages such as Lisp, Fortran, or COBOL. In fact, COBOL is a direct descendant of FLOW-MATIC and its main legacy. From the very beginning, the use of natural language in programming languages was associated with efforts to make programming more accessible to new audiences. In the words of Grace Hopper, the creator of FLOW-MATIC:



I used to be a mathematics professor. At that time I found there were a certain number of students who could not learn mathematics. I then was charged with "the job of making it easy for businessmen to use our computers. I found it was not a question of whether they could learn mathematics or not, but whether they would. [...] They said, 'Throw those symbols out

-- Grace Hopper

The Wikipedia article on FLOW-MATIC provides the following example program:

```
INPUT INVENTORY FILE-A PRICE FILE-B ; OUTPUT PRICED-INV FILE  
FILE-D ; HSP D .  
  
1  COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B) ; IF GREATER G  
   IF EQUAL GO TO OPERATION 5 ; OTHERWISE GO TO OPERATION 2  
2  TRANSFER A TO D .  
3  WRITE-ITEM D .  
4  JUMP TO OPERATION 8 .  
5  TRANSFER A TO C .  
6  MOVE UNIT-PRICE (B) TO UNIT-PRICE (C) .  
7  WRITE-ITEM C .  
8  READ-ITEM A ; IF END OF DATA GO TO OPERATION 14 .  
9  JUMP TO OPERATION 1 .  
10 READ-ITEM B ; IF END OF DATA GO TO OPERATION 12 .  
11 JUMP TO OPERATION 1 .  
12 SET OPERATION 9 TO GO TO OPERATION 2 .
```

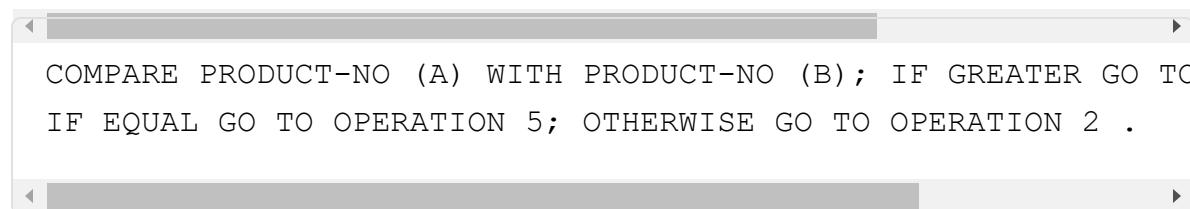
```
13 JUMP TO OPERATION 2 .  
14 TEST PRODUCT-NO (B) AGAINST ; IF EQUAL GO TO OPERATION 16  
    OTHERWISE GO TO OPERATION 15 .  
15 REWIND B .  
16 CLOSE-OUT FILES C ; D .  
17 STOP . (END)
```

An example program in FLOW-MATIC

Check your understanding: To what extent can you guess at the potential meaning of this program?

Discuss: What are your initial reactions to the program?

A common reaction upon first reading a program in FLOW-MATIC is to note its verbosity, even compared to unstructured, go-to-based code in later programming languages. Consider the following code fragment, which implements a comparison



The screenshot shows a window with two scroll bars, likely from a vintage computer interface. The text area contains the following FLOW-MATIC code:

```
COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B); IF GREATER GO TO  
IF EQUAL GO TO OPERATION 5; OTHERWISE GO TO OPERATION 2 .
```

FLOW-MATIC snippet: comparsion operation

This code snippet is roughly comparable to the following line of C code:

```
if(a>b) goto Op10; else if (a<b) goto Op5; else goto Op2;
```

The difference in the length of these two programs is substantial, and immediately evident upon looking at them; the FLOW-MATIC code is far more verbose than the C code. Nonetheless, a robust critique should carefully distinguish verbosity from complexity:

- * Verbosity is when a program expends a large amount of text to express a small amount of content.
- * Complexity can refer in a broader sense to the subjective difficulty of writing or comprehending a program, but here it refers in a narrow sense to the complexity of the formal grammar for a language's syntax. Complexity means that a language has a large number of keywords or a large number of rules in its grammar.

The example above argues the verbosity of FLOW-MATIC, but verbosity alone need not be a death-knell for a programming language. This is particularly true in the modern context, where robust tool support provides auto-completion as a programmer types their code.

Complexity makes a more compelling case against the use of a programming language: when a programming language has a large number of keywords or grammatical rules, it increases the cognitive load on programmers because a larger number of

distinct concepts must be kept in mind while reading or writing code. If a programmer must routinely consult documentation to remember syntax, it hinders productivity. The concern of complexity becomes particularly strong when evaluated FLOW-MATIC's descendant, COBOL. Published COBOL standards contain *hundreds* of distinct keywords, while modern professional PLs typically have a few *tens* of keywords. For languages such as COBOL, whose primary users are professional programmers who can invest time in learning syntax, the potential readability of natural language is vastly outweighed by the grammatical complexity of writing it.

In fairness to NLP, however, it is essential to inspect the motivations of the language designers, as with any language. From the earliest days, the motivations behind most NLP languages are psychological, not purely technical. Just as with Processing and Twine, a key design goal was to avoid intimidating programmers who have a fear of math. Whether NLP languages succeed on this point could only be assessed using methods from social science, not a simple analysis of the formal grammar.

This history lesson on FLOW-MATIC reveals the fundamental tension of designing NLP languages: the tension between professional programmers' desire for grammatical simplicity and a newcomer's desire for continuity with their experience of natural language. Next, we explore a modern programming

language where these tension resolves in a compelling way: fully-natural-language syntax which treats readability by non-programmers as a key feature, even if non-programmers do not succeed in writing it.

The language we explore next is Inform 7, or Inform for short, a domain-specific language for writing interactive fiction.

Inform

Inform is a programming language developed for a specific domain: interactive fiction games, particularly text adventure games. These games typically consist of an interactive world with which the programmer can interact by typing commands in a terminal, such as moving through the world, interacting with objects, or talking to non-player characters. Though the highly specialized domain of text adventure games limits the audience of Inform as a programming language, it is of broad interest as a case study in programming language design. The relationship between natural language and programming language syntax is relevant to all programming language designers, and Inform explores this relationship in a distinctive way, by developing a programming language syntax where programs are grammatical English sentences.

Though we do not discuss the Inform parser in detail, the reader should be aware that the emphasis on grammatical English sentences makes the parser substantially more complicated than most, because English grammar involves not only assessing parts of speech, but features such declension and conjugation, drawing in issues such as person, tense, and count. For example, Inform must understand that “walk” and “walking” refer to the same activity, and whether a noun X should be referred as “a X”, “an ”, “the X”, or just “X”.

Example: “Disenchantment Bay 1”

A major design goal for Inform is that its syntax should be readable even to people who have not studied how to program Inform. To evaluate that goal, we begin the study of Inform by reading a standard example program provided by the Inform developers, without prior explanation of the language.

The Cabin is a room.

"The front of the small cabin is entirely occupied with navi
a radar display, and radios for calling back to shore. Along
with faded blue vinyl cushions, which can be lifted to reve
underneath. A glass case against the wall contains several f
Scratched windows offer a view of the surrounding bay, and t
to the deck. A sign taped to one wall announces the menu of
Yakutat Charter Boat Company."

```
The Cabin contains a glass case. In the glass case is a coll  
The case is closed, transparent, and openable.  
The bench is in the cabin. On the bench are some blue vinyl  
The bench is enterable.  
Test me with "examine case / get rods / open case / get rods  
take cushions / get up"
```

Example program: Disenchantment Bay 1

Check your understanding: Before reading ahead, write a note to yourself explaining what you think this program does, in as much detail as possible. Without playing the game in Inform, attempt to figure out: what series of commands would you type, in what order, to obtain the fishing rods?

Discuss: What are your initial reactions upon reading this program?

One immediately visible aspect of Inform code is that it takes the natural language concept very seriously. In contrast to many other NLP languages, Inform code consists of grammatically-correct English sentences. The section between quotes, which is the description text of a room, is entirely free-form natural language text, to be displayed directly to players.

Can Inform code be read by non-programmers, or at least by programmers who have not been trained in Inform? If you

completed the check-your-understanding exercise, you will have an opportunity to assess that for yourself, by returning to the example in the exercises after reading the rest of this section.

Understanding Inform

This section does not attempt a formal definition of the syntax or semantics of Inform, because it is sufficiently complex that a formal definition would risk creating more confusion than clarity. Instead, this section organizes the key language concepts of Inform and provides example uses of key language features.

We will not attempt a remotely-exhaustive nor truly formal definition of Inform, because the language is too complex to make such an exercise realistic for one lecture. We will however organize many of Inform's key concepts. As in most programming languages, an Inform program consists of code and data, but its emphasis on building virtual worlds provides a unique twist on the concepts of code and data. The counterpart to typical code in Inform is writing gameplay rules that control the game world, and the counterpart to defining data is modeling the structure of the virtual world. Text in quotes serves as a free-form description of an object in the world, and text without quotes is used to define the world. This world is built up from components such Nouns, Actions, Kinds, Verbs, and Relations, all of which have formal

meaning in Inform. We first introduce the language features relating to world modeling, then discuss rules.

Modeling the World with Nouns

Check your understanding: List all the nouns that occur in the *Disenchantment Bay 1* example.

Answer: The nouns are “Cabin, glass case, fishing rods, bench, blue vinyl cushions”. Note that mentioning something in quotes does not make it a noun. For example, the description text in the example mentions a sign, but the sign is not defined as a noun.

Once we create a noun, we can assign additional meaning to it using a wide variety of words with special meaning in Inform, which can be viewed as a standard library. Words like “in”, “on”, “collection,” “some,” “closed,” “transparent,” “openable,” and “enterable” all have built-in, standard meanings which enrich the definition of a noun and how the player interacts with it.

Relations like “in” and “on” serve to construct a graph of in-game locations and other nouns, which can have the following physical relations to each other:

- * Adjacency: One location can be reached from another by traveling in the directions “east, west, north, south, up, down”

- * Support: One thing can be on or under another
- * Containment: one thing is inside another

By default, the standard library provides Inform games with a standard set of movement commands, like “go north,” “go up,” “enter *place*,” and so on. When the source code creates these physical relations between locations, the built-in movement commands let the player move between them without any added programming work. The “enterable” property relates to containment, indicating that the player can enter a location.

The other properties, such as “openable,” “closed,” and “transparent” affect other core aspects of gameplay. If an item is “openable,” it can switch between the states “open” and “closed,” and things can be removed from it when open. If something is transparent, any items inside it can be viewed by the player.

Controlling the World with Rules

We now explore the “computation” side of Inform: rules. Typical rules consist of an event under which the rule applies and an action that takes place upon application of the rule.

Example 1: “Report” rules. A reporting rule is applied after a given action takes place. The following code:

```
Report someone dancing:  
say "[The actor] dances a few steps of [the noun] for you."
```

Example: Reporting an Action

ensures that whenever someone dances, the above message will be displayed to the player.

Implicitly, this rule has two parameters, named “the actor” (the one doing the action) and “the noun” (the direct object of the action), which are referenced in square brackets. For example, if Aditi dances Viennese Waltz, the message would be “Aditi dances a few steps of Viennese Waltz for you”

Example 2: “At <time>” rules. Inform has a built-in notion of time, allowing a rule to activate when a given time passes.

The following code:

```
The time of day is 4:55 PM  
  
At 5PM:  
    now the Eastern Hemisphere is dark;  
    now the Western Hemisphere is lit;
```

Except from Hohmann Transfer example

Initializes the time-of-day to 4:55 PM and defines a rule which is activated once the time advances to 5pm. When that rule takes effect, the Eastern Hemisphere's lighting is set to "dark" and the Western Hemisphere's lighting is set to "lit".

Example 3: "Instead" rules. These are replacement rules which activate whenever a particular action would occur, and cause a different action to occur in its place.

The following code:

```
Instead of buying something:  
    say "You already have [a noun]"
```

Except from Introduction to Juggling example

Means that whenever the player attempts to use the "buy" action, the action does not take place at all, and instead a "say" action takes place, e.g., "You already have a doughnut" if the player attempts "buy doughnut".

Key Language Concepts

The Inform developers highlight the following language concepts besides Nouns: Actions, Kinds, Verbs, Relations. We discuss each concept in turn.

Actions

Actions are things the player can do. There are fundamental built-in actions for engaging with the senses, such as “look,” “listen,” “smell,” “touch,” and “taste”. Not every action is so directly tied to a single sense: moving from place to place, engaging in combat, conversing with non-player characters, and manipulating items are all actions. A programmer can also define their own actions.

These different actions share a few underlying traits from a programming language design perspective. Most actions are only applicable to specific nouns or specific kinds of nouns. For example, a player might be prohibited from looking at an abstract concept such as the natural numbers or barred from tasting non-player characters as a matter of basic human decency. Actions are capable of modifying the state of the game world, e.g., taking an object out of the container changes the state of the container and the player’s inventory. There are also common actions, such as looking, which typically do not modify the world.

Verbs

From a purely linguistic perspective, the name “Verb” can cause confusion when compared with the name “Action.” Surely, every action is a verb in the linguistic sense, but they describe only one class of verbs: *action verbs*. In contrast, Inform primarily uses the

term **Verb** to refer to *stative verbs*, which are used to the current (but mutable) state of the world. Nonetheless, because readers may associate the word “verb” with action verbs, we capitalize **Verb** to emphasize their specific role in Inform. Commonly-used Verbs in Inform including possession (“have”) and visibility (“[can] see”).

Verbs and Actions interact with one another: certain actions will only succeed if certain Verbs hold, and certain actions will change which Verbs hold. For example, if I *take* a cake (Action), the result is that I now *have* a cake (Verb). If I want to look at a cute dog (Action), I had better hope that I [can] see the dog (Verb), else the Action would fail.

Kinds

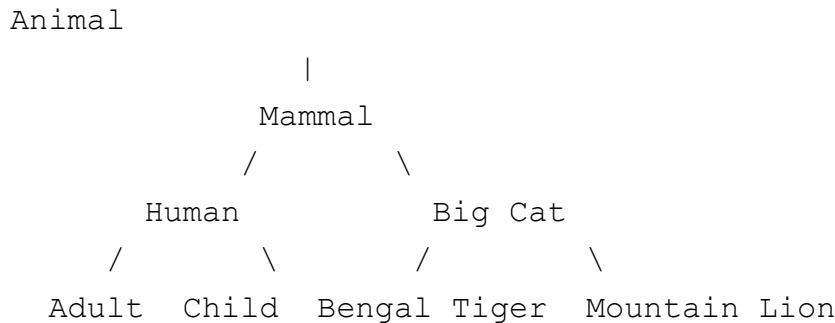
A challenge in defining Rules, Verbs and Actions is that most of them are applicable to multiple Nouns, but not every Noun. When defining Rules, Verbs and Actions, it is desirable to concisely specify which nouns they apply to. To solve this problem, Kinds serve as basic type system for Nouns. Specifically, the Kinds form a hierarchy, and subtyping is the essential relationship between different Kinds.

The following example program:

```
A mammal is a kind of animal.  
A human is a kind of mammal.  
A big cat is a kind of mammal.  
A Bengal Tiger is a kind of big cat.  
A Nittany Lion is a kind of big cat.  
An adult is a kind of human.  
A child is a kind of human.
```

Typing hierarchy example, from the author

Would build the following tree of kinds:



Tree depiction of type hierarchy

In definitions (e.g. of rules), we can specify which kinds the definition applies to. Imagine we develop a game that involves both humans and big cats that the player can interact with. If we define actions such as “attach leash” and “meow” to apply only to big cats, then Inform automatically uses this typing hierarchy to understand that these actions can be applied to

both a Bengal Tiger and a Mountain Lion, but not any Human such as an Adult or Child.

Note that Nouns can be defined using any kind from the hierarchy, not just the leaves. For example, Tony the Tiger might be defined directly as a Big Cat^a

^a Because Tony the Tiger is canonically Italian-American, it is unlikely that he is a Bengal Tiger specifically, thus he needs a broader classification. We place him directly under Big Cat because his connection to the species *Panthera tigris* is indirect at best. He is a fictional character.

Relations

On first sight, Relations appear similar to Verbs. They are distinguished by the fact that they specifically relate two or more entities to one another. Relationships we have encountered thus far include Adjacency, Support, and Containment. Other common examples include visibility, touchability, possession, wearing, carrying, and incorporation (X is a part of Y).

The “relates” keyword is used to define new relations. A provided example defines monogamous love:

Loving relates various people to one person.

Where multiple people can compete for love but cannot love multiple people at once.

Natural Language Processing and Programming

Separate from natural language programming, natural language *processing* is a substantial subfield of computing which address the computational treatment of human language. As a descendant of both linguistics and computer science, this field relies on a few technologies that are relevant to programming languages (such as formal grammars), but otherwise differs drastically in its goals and approaches. Typical approaches rely heavily on machine learning techniques applied to large corpuses of text. Problems of interest include translation, extracting structured data from unstructured text, processing commands for smart assistants, analyzing the sentiment of a text, and generating text for auto-completion or chatbots. The latter problem rose to exceptional public prominence following the release of ChatGPT (in late 2022) and other large language models (LLMs). GitHub Copilot (released in late 2021) is particularly notable because it is specialized in code generation. Due to this prominence, we discuss LLMs first, then other methods from natural language processing.

Large Language Models

The design of an LLM is wildly different from the design of a programming language. In an LLM, the primary design decisions

are problems of designing a machine learning architecture, not problems of defining syntax, semantics, and types, nor problems of identify programmer needs. Rather, LLMs are discussed in this book because they have been used to return programs when given natural-language prompts. This functionality has been used, e.g., to generate solutions to introductory programming problems or provide autocompletion.

As of this writing, commercially-available LLMs are distinguished from programming languages by their fundamental privacy and intellectual property issues, as they are trained using crawls of the Internet which include substantial amounts of material protected by copyright and private information, including legally-protected information such as medical records. Fundamentally, the design of an LLM is not analytical in nature, in the sense that its creators do not imbue it with any domain-specific knowledge, such as knowledge about programming. Rather, LLMs exploit massive quantities of data to develop strong word associations: given that the LLM sees a specific word in a user's query, it can make effective predictions about what words a reader would expect to see next, based on public Internet data.

The efficacy of LLMs on exercise-level programming problems can be explained in large part by the prevalence of educational materials and exercise solutions on the Internet. Efficacy decreases for large, complex programming tasks. In contrast,

programming languages with natural-language-based syntax scale for more easily to complex problems, because they are analytical in nature: every keyword in a natural-language programming language has a concrete meaning assigned to it by its designer. Even if efficacy were to improve over time, the ethical concerns with current-generation LLMs are too foundational to advocate their use as programming assistants.

Other Natural Language Processing Techniques

Despite the ethical concerns associated with LLMs, the field of natural-language processing encompasses many other technologies which have been successfully applied to ethically-obtained data, some of which have potential applications to programming languages:

- * Machine-learning-based tools have been developed which identify when a natural-language document violates the grammatical rules of the given natural language. Such tools could be explored as error-message-generators for programming languages with a formally-defined natural language syntax, helping provide clearer communication than a traditional parser error message
- * Sentiment analysis could be applied to programs, including their comments. This analysis could be used to detect programmer frustration and encourage a solution to their

frustration, or to detect text which violates community rules, e.g., against hate speech

Discussion

As of this writing, NLP is substantially less widespread than styles of programming language syntax. This chapter does not go as far as to advocate that NLP should be made widespread, and certainly does not advocate that every programmer needs to become proficient in the specific domain of text adventure programming. Instead, this chapter aimed to explore NLP languages in order to reinforce broader themes of the book and extract generalizable insight about the design of programming languages. We reinforced the following themes:

- * To design a programming language, we must understand the person we design it for and their goals. For example, developing an operating system in Inform would likely be a struggle, but Inform should not be blamed as it is not designed for that task
- * Continuity across languages is an important design consideration, which can be manifested in different ways. The design of Inform prioritizes continuity with written English text for a reader (not writer) who is comfortable with English prose

We close the discussion with reflections from the author:

- * A human-centric perspective on programming language design should not consider a universal notion of what makes a programming language “usable”. Instead, the design goals of a programming language must be determined by assessing the wishes of the target audience.
- * The human-centered aspect of programming language design is not an added burden, but an opportunity for significant innovation in language design, including innovative syntax and tooling.
- * Programming language design is something that continues to occur in the modern day, not only in history. While the most popular languages for professional programmers have long histories, new languages continue to emerge, both for specialized and general-case use.
- * The history of programming languages is important not only for fans of history, but for all programming language designers, because certain major ideas in programming language design continually reoccur throughout the years. When designing a new programming language whose inspirations coincide with an older language, the limitations of the old must inform the design of the new.

TODO

Create nice figure for typing tree

Related Work

Subtyping

ources: Inform (7) was developed by Emily Short and Graham Nelson, a wife-and-husband team. This lecture is largely based on their work, especially https://ganelson.github.io/inform-website/book/WI_1_1.html

Literate Programming

Classroom Activities

- * Prompt students for their initial reactions to programs in each language
- * Play the example Inform game “Desolation Bay” in class.
- * Invite students to share their reflections on the case studies.

Exercises

- * Kind hierarchies and other definitions such as Relation definitions in Inform are structuralist, they define the objects of the world in neatly-divided and exhaustive categories. Explore how these hierarchies can reflect the cultural norms of their authors. Identify or write a definition that asserts such a norm and then write a definition that asserts an alternative to that norm. This chapter presented several definitions which could be used for this exercise
- * Write an Inform program which defines a kind hierarchy containing your favorite animal and its relatives
- * This book discusses two tools used for interactive fiction: Twine and Inform. Compare and contrast them in any *one* of the following ways:
 - * Read an example program for either tool and translate it into the other. Report briefly on your experience.
 - * Research project: Try to identify a class of formal automata that correspond to Inform programs. Compare the expressive power of that class to DFAs, which correspond to Twine programs.
 - * Research project: Implement a sentiment analysis tool which detects programmer frustration
 - * Grace Hopper is featured in this chapter because she developed the first natural programming languages. This

contrasts with a common reason she is included in computer science educational materials, as a (sometimes tokenistic) example of a woman in the history of computer science.

Though Grace Hopper has full claim to that identity, she is not an intersectional figure, and excessive cultural focus on Hopper risks minimizing intersectional figures in computer science. Pick such a figure in computer science, research their life, and write a one-page biography about them

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter21

Chapter 18

Diagramming

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

How can you create visually compelling figures about complex technical topics without substantial experience in graphic design? How might a programming language assist you in that effort? This chapter explores Penrose, a domain-specific language for mathematical diagrams, which aims to answer these questions. The development of Penrose can be motivated by the following quote:



Mathematicians usually have fewer and poorer figures

in their papers and books than in their heads.

-- William Thurston

Mathematicians and practitioners of other STEM disciplines are knowledge workers, creating new knowledge which deserves to be disseminated. The impact of a new technical result is only proportional to how effectively it is communicated. However, most experts in STEM disciplines have no formal training in graphic design or the typesetting of mathematical figures, which can hinder the communication of new technical knowledge because figures are essential communication aids. Though STEM workers typically lack formal graphic design and figure typesetting training, they are typically comfort with equations, code, or both, thus a programming language is a plausible tool to assist this community with developing figures.

Penrose is a domain-specific programming language designed to help mathematicians and other STEM workers generate visually appealing diagrams. Penrose is designed around the separation between abstract mathematical definitions and their concrete visual representation, with the goal of supporting:

- * Separation between content and presentation
- * Accessibility toward new users

- * Evolvability of code over time
- * Scalability to large programs

Example Program

We begin our exploration of Penrose by reading an example program from the original article on Penrose:

```
Point p,q,r,s
Segment a := {p,q}
Segment b := {p,r}
Point m := Midpoint(a)
Angle theta := ∠(q,p,r)
Triangle t := {p,r,s}
Ray w := Bisector(theta)
Ray h := PerpendicularBisector(a)
```

Example Penrose program

The first line is a *declaration*, which states that four points exist, but allows them to be arbitrary points. All the remaining lines are *definitions*, indicated by the assignment symbol (written `:=`). Respectively, the remaining lines create two line segments, a midpoint, an angle, a triangle, and two rays, all of which collectively make up the final diagram. This diagram is not particularly meaningful in mathematics, but serves to demonstrate common features.

It is important to note that definitions in Penrose are radically different from those in most programming languages. In most programming languages, every declaration must eventually be implemented by some definition. In Penrose, however, a declaration can be left completely undefined, and that undefined declaration can even be used to build up more complex structures in following lines. This flexibility illustrates why Penrose can be classified as a declarative language: it focuses on specifying a problem (the content of a diagram to be laid out in a visualization) but not a full solution (full presentation information for the diagram). Instead, the Penrose implementation takes responsibility for inferring the values of undefined values, through which it also infers presentation.

A consequence of making the implementation responsible for presentation is that it becomes possible to radically change the presentation of a diagram without any revisions to the Penrose program that defines the diagram's content. For instance, the example figure can be rendered in 3 different styles corresponding to different sets of geometric axioms: Euclidean geometry, spherical geometry, and hyperbolic geometry.

Syntax: Types, Predicates, Functions

Before we can answer what a program means, we need to answer: what is the syntax of programs? Penrose content roughly corresponds to simply-typed first-order logic.

“Simply-typed” has a precise technical meaning, and is not about being simple. “Simple typing” means that types are neither polymorphic (no type parameters like `List<Int>`) nor dependent (no term parameters such as `IntVec<5>` for vectors of 5 integers).

Penrose supports notions such as “Set” and “Point” but not “Set of points” nor “Set of three points”. First-order means that quantifiers are allowed, so that we can make statements about “all” or “some” element of a type.

Similar to first-order logic, the core concepts in Penrose expressions are functions and predicates. Functions transform values into values, such as computing an angle out of several points. Predicates transform values into Boolean truth values `true` and `false`, such as the equality predicate (`=`) which is true exactly when its two arguments are the same. For our purposes, this is all the Penrose syntax we need to know. For using Penrose, you would want to know some of the advanced syntax from the paper and also know its standard library, which is substantial.

The concrete syntax depicted in the paper is parsed and processed into a computation graph format. A computation graph is like an abstract syntax tree except that it need only be a directed graph and not a tree. For example, if the same variable appears twice in a given expression, both occurrences will be represented with a single shared node instead of two independent branches.

Semantics: Constraints

When we want to know “what does the program do?” we typically look to the program semantics for answers. Penrose does not define a formal mathematical semantics such as an operational semantics, only an informal semantics. Even the informal semantics is insightful, however.

The semantics of a program is a first order constrained optimization problem. Constrained optimization problems arise everywhere throughout computer science; though especially widespread in artificial intelligence, they arise even in programming languages. In their most basic form, every constrained optimization problem consists of three parts: variables, constraints, and an optimization objective:

- * **Variables:** A solution to the problem consists of values for each variable

- * **Constraints:** A solution must make all constraints true at the same time
- * **Objective:** An optimal solution *should* make the objective value as optimal (either small or large, depending on the specific problem) as possible without breaking the constraints.

Constraints and objectives come from the *style* language, not the *content* language, and we have not yet discussed the style language. In short, it has two keywords “ensure” and “encourage,” which respectively create constraints and objectives.

Constrained Optimization Example:

Suppose we have a diagram of a tree, consisting of circles depicting each node, where we seek a compact but non-overlapping diagram. Then, the components of the constrained optimization problem are:

- * **Variables:** Every basic graphical element in the diagram
- * **Constraint:** No two circles overlap
- * **Optimization goal:** minimize the size of the diagram.

The use of constrained optimization relies on the principle that there will exist optimization problems that align with what

humans really want from diagrams. It is *not* to say that *everything* we want from diagrams can be reduced to optimization. Of course, humans care about aesthetics in ways that don't reduce clearly to objectives. However, some of the core problems in diagramming do fit this framework.

- * **Layout:** This is arguably the single most compelling application of constrained optimization for diagrams. Optimized layouts typically minimize space

usage or maximize alignment between different elements while preventing unwanted overlaps

and/or ensuring minimum font sizes

- * **Colorization:** Another key application of constrained optimization is to determine the coloring of a diagram. Color schemes can be picked using different constraints and goals to maximize

contrast, maximize obedience of color-theoretic laws, or accommodate colorblindness

Layout and colorization need not be the only applications of constrained optimization for diagrams, but they are substantial applications in their own right. These applications of optimization become even more useful if they are combined with a limited

degree of human agency, e.g., to pick a desired layout from among several options. Integrating this agency in Penrose is straightforward because it does not compute a globally optimal solution, instead it computes locally-optimal solutions from randomized starting states using a family of methods known as exterior-point methods.

Style Language

An advantage of providing separate content and style languages is that each language can target a different programmer audience. The style language is meant to be used by a smaller group of developers with greater expertise, because each style is intended to be reused across many content files.

The basic structure of a style program is a series of rules of the form:

```
forall Type t {  
    t.field = expression;  
}
```

The basic structure of a style rule in Penrose

where the clause `forall Type t` is called a *selector* and the clause `t.field = expression` is

called a *declaration*. In addition to this basic form, the selector section can contain a clause

`where predicate(t)` which restricts the selector to values `t` of the given Type which satisfy the

given predicate. In addition to this basic form, the declaration section can include clauses

`encourage predicate` and `ensure predicate` which respectively generate objectives and constraints.

In simple terms, a selector decides “which things should I give style to?” and the declaration decides

“how should I style them?”

We give an example rule from the Penrose paper. The rule, which generates presentation information for vectors, is edited for simplicity, and has end-of-line comments indicated by double hyphens (`--`).

```
forall Vector u, VectorSpace U -- match any vector
  where In(u, U) { -- in some vector space
    u.arrow = Arrow {
      startX : U.originX
      startY : U.originY
      endX : ?
      endY : ?
      color : Colors.mediumBlue
    }
  }
```

Penrose rule, Vector example

This example says that for vectors $u \in U$ (for any U), we define the vector's arrow by:

- * copying the start coordinates from the vector space's origin
- * telling the optimizer to solve for the end coordinates (this is what means)
- * setting the color to medium blue.

Depending on which programming languages you know, the style language could seem familiar (e.g. from CSS). We now get some practice with Penrose syntax and then move on to discussion of design insights.

Practice: Chemistry

Penrose comes with support for several different STEM domains, including chemistry diagrams, specifically Lewis structure diagrams for chemical compounds. To generate a Lewis structure diagram, first define the atoms in the diagram, then the bonds between them, then the number of valence electrons remaining for each. We present the official example code for a Lewis structure diagram of nitric acid:

```
Hydrogen h
Nitrogen n
Oxygen o1, o2, o3
-- bonds
Bond b1 := MakeSingleBond(h, o1)
Bond b2 := MakeSingleBond(o1, n)
Bond b3 := MakeDoubleBond(n, o2)
Bond b4 := MakeSingleBond(o3, n)
-- electrons
ZeroValenceElectrons(h)
ZeroValenceElectrons(n)
FourValenceElectrons(o1)
FourValenceElectrons(o2)
SixValenceElectrons(o3)
```

Penrose example: Lewis structure diagram of nitric acid

The first three lines indicate that the compound contains one nitrogen atom, one hydrogen atom, and three oxygen atoms,

consistent with nitric acid's chemical formula NHO_3 . The next four non-comment lines establish the bonds (drawn as lines) between various pairs of atoms and the final five non-comment lines establish the remaining valence electrons (drawn as dots) of each atom.

For readers seeking additional practice with Penrose, consider writing programs that generate the following Lewis structure diagrams for other compounds:

Diatomc Oxygen

:O::=O:

Diatomc Chlorine

::Cl:-:Cl::

Nitrate:

:O:=N-:O::

|

:O::

Water:

H-O-H

Formaldehyde

:O:

||

H-C-H

Examples of Lewis Structure diagrams

Connections to Other Languages

We connect ideas from the design of Penrose to other programming languages, both high and low-level.

Cascading Style Sheets (CSS)

The Penrose style language is heavily inspired by CSS, which is the standard language for styling on the Web. The separation between content and presentation is such a key goal of modern HTML and CSS (respectively) that they are quite likely an inspiration for the content-presentation separation in Penrose. CSS should not be blindly admired for being a standard, but rather contains several core design insights that have stood the test of time:

- * Styling is typically applied to entire batches of elements. Consistency is a core element of style, so it typical, e.g., to want all elements of a list to display similarly, or all vectors, in a diagram. These batches need not always align directly with an element type (such as “list item” or “vector”), but can be tagged as “classes” when needed. Thus, the notion of selectors is core to CSS. A selector determines which batch of elements styling is applied to.
- * Styling is not monolithic. An individual element may have a huge variety of style attributes, which may be determined from different sources. In CSS, *cascading* refers to the ability

to integrate styling from those different sources. As a typical example, a single webpage might:

- * Leave the margins unspecified, taking default values from the web browser
- * Specify a certain justification style for all headings
- * Set a special font for the heading used for the title of the page
- * Styling is declarative, not imperative. Though the word “declarative” is often ill-defined in programming language design, its meaning here is clear; semantically, a CSS declaration does not perform mutation, CSS has no notion of state, and stateful reasoning is not required to reason about CSS code.

Constraint-Satisfaction Programming (CSP)

The first time you see constraint-solvers used in programming, they can seem like magic. However, the use of constraint-solving in Penrose is but one instance of a general category of programming languages called Constraint-Satisfaction Programming (CSP) languages, generally considered one kind of *declarative* language.

Informally, the distinction between CSP programs and traditional programs is that most (imperative and functional) languages are

about programming algorithms, about programming *solutions* to problems, yet CSP programming is about coding up *statements* of problems. As a rule, problem statements are typically shorter than solutions, leading to concise CSP code. Yet there is no free lunch. In CSP, the challenge is to state problems that are solvable by the programming language implementation. There is a whole subfield of algorithms devoted to such languages, many of which are simple enough that one hesitates to call them programming languages, e.g.:

- * Linear programs
- * Mixed integer linear programs
- * Quadratic programs
- * Logic programs

CSPs are widely used. Linear programs and their cousins are widely used in

AI and optimization; logic programs are widely used in experimental academic programming languages,

and CSPs in general are widely used in procedural generation of digital media.

Bytecode Languages

Though the strictest notion of a compiler is a program that translates source code into machine code executable by hardware, there are many compilers which target other languages. If the target language is high-level, the compiler is usually called a transpiler. Many of these target languages are not high-level, however, as it is also common to target machine-independent low-level languages known as *bytecode languages*.

In industrial use, the most widespread bytecode languages are the Java virtual machine (JVM) and .NET bytecode; academic counterparts include typed assembly language (TAL). These are abstract, portable languages which are suitable either for rapid interpretation or for just-in-time (JIT) compilation to an executable machine language. What do bytecode languages have in common with Penrose?

The common thread is that Penrose also uses a multi-stage compilation process, and the output of its compilation process is not machine code. Instead, Penrose programs are compiled by *generating* a constrained optimization problem from the source program. After that compilation finishes, the optimizer is invoked to create the final result. In summary, compilers can be created to and from a wide array of languages.

Discussion of Design Values

We close with a reflection on the design values of Penrose as they manifested throughout the chapter, and how they may differ from Processing, the other visually-focused programming language discussed in this book:

- * **Separation of concerns:** In Penrose, we reuse content and style code by clearly separating those concerns from each other. In contrast, Processing encourages reuse by copying and remixing existing code.
- * **(Set) Abstraction:** Styling is applied to entire sets of elements using selectors. This is abstract in the sense that we do not know which elements will be selected when we write the selector. In contrast, Processing prioritizes the concrete over abstract.

The values of separation of concerns and set abstraction are not shared between Penrose and Processing. These differences should be surprising: the intended users for Penrose are mathematicians and other STEM professionals, all groups which are typically comfortable working with abstraction and employing separation of concerns, which are not traditional design values in the media arts. Compared to Processing, the design values behind Penrose are more closely aligned with CSS, as both languages seek to enable technically-trained users to develop visuals.

Classroom Activities

- * Explore the following official examples together in the Penrose web interface:
 - * Euclidean Geometry
 - * Lewis Structure of Nitric Acid
 - * Sets as Venn Diagrams in 2.5D
- * Watch the constrained optimization process unfold in real time
- * In cases where the constrained optimization process gives visually unappealing results, explore potential sources of the issue
- * Make customization to style files and observe the impact on a given diagram

Exercises

- * Write Penrose programs to produce the example Lewis structure diagrams from this chapter
- * Open any textbook and attempt to recreate one of the diagrams from it in Penrose
- * Research every common variety of colorblindness. Create Penrose style sheets that accommodate each.

- * A common limitation of CSS in practice is that the separation of content and style frequently breaks down, with HTML and CSS files becoming tightly coupled with one another. Research this phenomenon and identify an example of it. Based on a critique of that example, assess the extent to which you do or do not expect the same breakdown of content-style separation to occur in Penrose.

Related Work

Reading: <https://dl.acm.org/doi/pdf/10.1145/3386569.3392375>

Tutorial: <https://penrose.cs.cmu.edu/docs/tutorial/welcome>

Try it in your browser: <https://penrose.cs.cmu.edu/try/>

Exterior point methods

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter16

Chapter 19

Process Calculus

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

How do two programs communicate with one another? This question is a fundamental one in the development of concurrent software, consisting of multiple processes running side-by-side, which communicate and interact with one another. These programs could run collectively on a single processor core (interleaving concurrency) or on separate processors, even separate devices (distributed programming). Because concurrent programs are notoriously hard to get right, they have become a focus of substantial programming language design effort, both from theoretical and applied perspectives.

The Theorists main tool for studying concurrent programs is Process Calculus, a wide family of formal programming languages for expressing computations which unfold through communications between processes. In defining (a) Process Calculus, we will use familiar approaches to defining Operational Semantics and Types, albeit in a more complex manner than we have used previously.

This chapter explores a specific kind of Process Calculus called Pi Calculus (PC). We explore PC because:

- * Though its operators may behave subtly, it can be defined using a relatively small number of operators,
- * Despite this relative simplicity, it is expressive: PC allows higher-order communications that send communication channels over other channels, through which it achieves Turing-completeness, and
- * Due to this combination of simplicity and expressive power, PC is a major influence on other Process Calculus languages that followed it. Once you gain proficiency with PC, other Process Calculus languages should be substantially easier to learn.

Defining Pi Calculus

We first present the syntax of Pi Calculus (PC), then explore its behavior on specific examples, then present its operational semantics.

Syntax

The most basic programs in PC are *names*, which we write using the identifiers u,v,w,x,y, and z. The behavior of names will prove to be fundamentally different from the mutable and immutable variables seen so far. Specifically, we will be able to generate new names dynamically and test whether two names are the same. We should expect that handling of “something new” is a necessity for distributed programs, because when a first subprogram receives data from a second, it could be receiving data it’s never seen before.

A program in PC is made of processes (variables: P, Q, R) which are named by process IDs (variables: A, B, C).

The language of processes is defined by

$$P, Q, R \leftarrow 0 \mid P_1 + P_2 \mid y(x).P \mid \bar{y}x.P \mid t.P \mid (P_1 \mid P_2) \mid (x)P \mid [x=y]P \mid A(y_1, \dots, y_n)$$

The syntax of PC is particularly challenging, even for experienced programmers. We walk through the syntax of PC case-by-case.

Sums 0 and P1 + P2

The program 0, called *inaction*, is the program that cannot do anything at all. It is the unit of the addition operator, i.e., $0 + P$ is the same as P for all processes P .

The symbol “+” should be understood as “or”. The program $P_1 + P_2$ either runs P_1 or runs P_2 . This fact that a program might do two different things when we run it is called *nondeterminism*, and is a key characteristic of PC. In contrast to traditional sequential programs, it is expected that the execution of a concurrent program will unfold in multiple potential ways depending, e.g., on scheduler behavior.

Prefix forms y(x).P and ŷx.P and t.P

In PC, the heart of computation is communication between processes, which is achieved using three programs collectively called the prefix forms, so named because each of them does one step of communication as a prefix, and then runs the rest of the program (written P).

In the positive prefix form $y(x).P$ and negative prefix form $\bar{y}x.P$, name y is respectively understood as an input port or an output port for the process. Process $y(x).P$ reads a value (names are values, and the only values that we will really discuss in this chapter!) from y and binds that value to x . Program P can refer to x . In contrast, $\bar{y}x.P$ writes the value of x to y , then runs P . It does not bind any names.

In process calculus, “reading y ” and “writing y ” are both called actions, and there is traditionally a do-nothing action called the silent action, which represents local computations in a process with no communication. The silent prefix form $t.P$ represents doing the silent action, then P .

Composition P1 | P2

The composition $P1 | P2$ should not be confused with the sum $P1 + P2$. Composition $P1 | P2$ means that both processes $P1$ and $P2$ are running in parallel. There are two kinds of execution steps the program $P1 | P2$ can take. Firstly, the subprocesses $P1$ and $P2$ can synchronize, i.e., communicate with each other by reading and writing to the same channel. When they do so, this computation is considered local to $P1 | P2$, i.e., the overall process $P1 | P2$ takes the silent action because the input and output cancel out. The second kind of step is for a single subprocess to take a step while

the other does nothing. When P1 or P2 takes such a step. Its action is also considered to be the action taken by $P1 \mid P2$.

Restriction

The restriction $(x)P$ is like P, but it cannot take an input or output action at name x. Notably, P can perform internal communications on x. For example, if $P = P1 \mid P2$, then P1 could write a message to x which is read by P2. The restriction is only against communications which are externally visible. Though this definition makes it sound like restriction deletes a name x, the common intuition is just the opposite: because restriction ensures x cannot be read or written externally, we think of it as creating a new local name out of nowhere.

Matching

The matching process $[x=y]P$ tests whether the names x and y are the same or not. If so, it runs process P, else it runs process 0 (the process that cannot do anything). This is how conditional statements are implemented in PC: the process $[x=y]P$ can be read as an “if” statement $if(x=y)P$, with no “else” branch

Named Processes

We assume there exists some way to define named processes, which can have parameters and which can be recursive, e.g., definitions might have the syntax:

$A(x_1, \dots, x_n) \equiv P$ where P can mention x_1, \dots, x_n, A .

This definition mechanism does not appear explicitly in the process syntax. But, once a process $A(x_1, \dots, x_n)$ is defined, it can be instantiated using function call syntax: $A(y_1, \dots, y_n)$.

Example Code

We explore several short, toy PC expressions to gain familiarity with the syntax.

Send One Message

The following example sends a message along a channel named a by using the send command $a(b)$ and receive command $\bar{a}c$ on parallel branches:

$(a)(b)(a(b).0 \mid \bar{a}c.0)$

The example first defines names a and b . Name a is used as a channel, while name b (which, like every name, is a value) is sent across the channel a using the syntax. In parallel, the second process executes the read command $\bar{a}c$ which binds the new name c to the value received over channel a , which is b . After completing this interaction, both processes continue on the empty process 0 , i.e., they stop.

Link passing

Consider 3 agents P, R, Q . In this scenario, P has a link to R , named x . Our goal is to pass link x to Q over a link named y . We define a PC program which accurately models this scenario. Note that the scenario puts no requirements whatsoever on R , so we need not place requirements on it either.

Let P be $\bar{y}x.0$, let Q be $y(z).0$, and R be an arbitrary process. Then the program

$P \mid Q \mid R$

passes the link x over the link y .

Scope Intrusion

Variable scope management is one of the major technical subtleties of process calculus, because sending links between processes requires careful management of variable scope (i.e., where the variable is defined or not). Scope Intrusion and Scope Extrusion are two sides of this the same coin.

For scope intrusion, consider four processes P,Q,R,S. Suppose that Q has a private link to S and P has a private link to R, and that both are named x. P has another link y to Q, and wants to send its x across link y.

That is, let P be $\bar{y}x.0$, Q be $(y(z).0)$, and R and S be arbitrary, then consider the program $(P \mid R \mid ((x)(Q \mid S)))$

In this case, we say (the communication from) P *intrudes* on the scope of the private link x between Q and S. Intrusion is resolved by (automatically) renaming the private variable to a fresh variable, e.g., to

$$(P \mid R \mid ((z)(Q\{x/z\} \mid S\{x/z\})))$$

where the notation $\{x/z\}$ indicates renaming every z to x.

Scope Extrusion

This example is like the Link Passing example, except the link x between P and R (which is sent to Q over y) is private. That is, P be $\bar{y}x.0$, let Q be $y(z).0$, and R be an arbitrary process. Consider the program $((x)(P \mid R) \mid Q)$. After one step of execution, this will become $(x)(0 \mid R \mid 0\{x/z\})$

Because x is private to P and R , it is renamed to a fresh name z in Q , so that Q can never access it. The scope of x is extruded (expanded) so that x is now bound at the top level, i.e., it is no longer private to P and R .

Semantics

The are several ways to develop the semantics of processes. As we have hinted in the examples, one way to understand the meaning of processes is through an operational semantics, i.e., which programs step to which other programs. Operational semantics does “work” for process calculus, but for several reasons, it is not the semantics we introduce here:

1. Stepping is a very strong relationship. There are many programs that mean the same thing but do not step to each other, and

2. Processes are highly nondeterministic, and a single program might step to many others, leading to combinatorial explosion in complexity of operational reasoning.

Instead, process calculus typically focuses on equational reasoning: when do two processes mean the same thing? We skip the formal definition of “mean the same thing” because it is fairly technical, but look up “strong bisimilarity” for more information. In the following rules, let $P = Q$ mean that P and Q are strongly bisimilar; let $P =_a Q$ mean that P and Q are alpha-equivalent (differ only by bound variable naming), let $A(x_1, \dots, x_n) \equiv P$ mean that $A(x_1, \dots, x_n)$ is defined to be P , and let variables α, β stand for “prefixes”, i.e., the beginnings of prefix forms.

Then the semantics of PC consists of the following rules. The rule names we use are from Milner, et. al’s “A Calculus of Mobile Processes”

Rule A

$$P =_a Q$$

$$P = Q$$

Rule A says processes are “the same” when they “differ only in bound naming”.

The next group of rules, the C0 rules, are congruence rules. These rules indicate that two equal processes remain equal when inserted as subprograms of the same surrounding code.

Rule C0a

$$P = Q$$

$$t.P = t.Q$$

Rule C0b

$$P = Q$$

$$\bar{x}y.P = \bar{x}y.Q$$

Rule C0c

$$P = Q$$

$$P + R = Q + R$$

Rule C0d

 $P = Q$

 $P \mid R = Q \mid R$

Rule C0e

 $P = Q$

 $(x)P = (x)Q$

Rule C0f

 $P = Q$

 $[x=y]P = [x=y]Q$

The last remaining congruence rule, C1, indicates that equal programs remain equal when placed under the same receive command, but that renaming is required.

Rule C1

 $P\{z/y\} = Q\{z/y\}$

$$x(y).P = x(y).Q$$

(where z is a fresh variable, occurring nowhere in P or Q)

The following rules characterize summation:

- * S0 means 0 is the unit of +
- * S1 means self-addition cancels
- * S2 means + is commutative
- * S3 means + is associative

Rule S0

*

$$P + 0 = P$$

Rule S1

*

$$P + P = P$$

Rule S2

*

$$P+Q = Q+P$$

Rule S3

*

$$P+(Q+R) = (P+Q)+R$$

¶

The following rules characterize restriction :

- * R0 deletes a restriction if not needed
- * R1 swaps order of restrictions
- * R2 distributes restriction over addition
- * R3 commutes restriction with prefixes when safe to
- * R4 detects restricted prefixes and equates them to 0

Rule R0

*

$$(x)P = P$$

(if x not a free name of P)

Rule R1

*

$$(x)(y)P = (y)(x)P$$

Rule R2

*

$$(x)(P+Q) = (x)P + (x)Q$$

Rule R3

*

$$(x)\alpha.P = \alpha.(x)P$$

(where x is not in prefix α)

Rule R4

*

$$(x)\alpha.P = 0$$

(where x is in prefix α)

The following rules characterize matching

Rule M0

*

$$[x=y]P = 0$$

(where x and y are distinct)

Rule M1

*

$$[x=x]P = P$$

Collectively, rules R0 and R1 mean $[x=y]P$ behaves as P when the test $x=y$ succeeds, and behaves as the empty process 0 otherwise.

The remaining rules are for expansion (E) and identifiers (I). The expansion rule is the most sophisticated we've seen yet. It assumes P and Q are organized as sums of prefix forms.

Rule E

$$P = \alpha_1.P_1 + \dots + \alpha_n.P_n$$

$$Q = \beta_1.Q_1 + \dots + \beta_m.Q_m$$

$$P \mid Q =$$

$$(\alpha_1.(P_1 \mid Q) + \dots + \alpha_n.(P_n \mid Q))$$

$$+ (\beta_1.(P \mid Q_1) + \dots + \beta_m.(P \mid Q_m))$$

$$+ (t.R_{ij} \text{ for all } i \text{ and } j \text{ such that } \alpha_i \text{ complements } \beta_j)$$

(where the α s bind no free names of P, the β s bind no free names of Q, and where the relationship α_i complements β_j holds in the following cases, each with its corresponding definition of R_{ij} :

- 1.** R_{ij} is $P_i \mid Q_j \{u/v\}$ when α_i is $\bar{x}u$ and β_j is $x(v)$

- 2.** R_{ij} is $(w)(P_i\{w/u\} \mid Q_j\{w/v\})$ for fresh w when α_i is $\bar{x}(u)$ and β_j is $x(v)$
- 3.** R_{ij} is $P_i\{u/v\} \mid Q_j$ when α_i is $x(v)$ and β_j is $\bar{x}u$
- 4.** R_{ij} is $(w)(P_i\{w/v\} \mid Q_j\{w/u\})$ for fresh w when α_i is $x(v)$ and β_j is $\bar{x}(u)$.

The high-level goal of rule E is to expand a composition-of-sums into a sum-of-compositions, which might have quadratically-many terms. The first two summands in the rule capture the simpler aspect of composition, where $P \mid Q$ could act as a sum where each P_i runs in parallel with Q or each Q_i runs in parallel with P .

The “complement” terms capture more complex interactions, specifically they use variable binding to capture what happens when one branch of $P \mid Q$ binds a variable in parallel with the other.

Rule I

$$A(x_1, \dots, x_n) \equiv P$$

$$Q(y_1, \dots, y_n) = P\{y_1/x_1, \dots, y_n/x_n\}$$

Rule I says that when a named process is encountered, it should be expanded by substituting the arguments into its body. Recall that

$A(x_1, \dots, x_n) \equiv P$ means $A(x_1, \dots, x_n)$ is defined to be P .

Conclusion

Pi calculus (PC) is one of the main versions of process calculus, a family of programming languages for modeling concurrent programs. Process calculi formalize programs as systems of processes which can execute concurrently, using communication between themselves for synchronization, using names for their connections, and potentially sending connections to each other, providing mobility.

We explored an equational semantics for pi calculus. We did not discuss its soundness proof, which would proceed by proving the equational semantics had an equivalent meaning to some other version of semantics. That semantics would typically define programs by their externally-visible communications; two programs may differ as much as they want in their external behavior, but be considered equivalent so long as their external interactions are identical.

Related Work

A Calculus of Mobile Processes

TODO

Talk about free and bound names much earlier on

More pretty pictures

TODO more better example before technology.

link passing example is not compelling

more better integrated into book

Look for an implementation they can play with

Classroom Activities

- * Draw circle-and-arrow diagrams for the example PC programs
- * Draw execution traces which walk through execution step-by-step

Exercises

- * In the language of your choice, implement a datatype representing ASTs for Pi Calculus

- * Design and implement an algorithm that attempts to determine whether two processes are equivalent. This algorithm should be based on the equivalence rules provided for process calculus. Because equivalence of two processes is undecidable, your program will not be able to determine equivalence in all cases. Either write an algorithm which loops forever when equivalence cannot be determined, or return a value “unknown” when you cannot determine equivalence.
 - * Write example processes. Write enough distinct processes so that your algorithm can successfully compare some for equality, but fails on others.
 - * What kinds of problems does your algorithm work on? What kinds of problems does it not work on?
 - * Which rules contributed to cases where the program loops forever or returns “unknown”?
- * Go is a programming language used for programming distributed systems in production. Read an introductory article on Go and then write a short summary of its differences and similarities when compared to Pi Calculus. How do their goals, approaches, and results compare?
- * Project: Design a small-step operational semantics for Pi Calculus. Prove that it is equivalent to the equational semantics
- * Implement the following in Pi Calculus:

- * A counter process which keeps track of a natural number: it can be sent messages which ask whether the number is zero, increment it, or decrement it
- * A collection of processes which implement lists. Implement an empty list as a process, as well *cons* process, which serves to add an element at the front of a list
- * An *identity* process which, given another process, forwards all communications to and from the process. For this exercise, you will want to assume the process follows some specific protocol, such as every receive being followed immediately by a send
- * A bank account. It should accept commands to increase and decrease its balance by an arbitrary amount (in contrast to the counter) and only allow money to be withdrawn if the balance is sufficiently high. It will likely requiring writing “helper processes” corresponding to helper functions.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter17

Chapter 20

Cost Semantics

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

- * Sequential cost semantics
- * Parallel cost semantics for fork-join parallelism
- * Parallel cost semantics for data-parallelism

Time is everything, or so the saying goes. This saying certainly holds true when it comes to writing software. As Moore's law slows down and the performance of computing hardware begins to reach its limits, the computational demands of software have only grown greater:

- * Scientists everywhere from physics to climatology find themselves needing to interpret larger and larger sets of experimental data,
- * Machine-learning algorithms, with a growing array of applications, continue to demand ever-larger amounts of compute time, hardware, and electricity, and
- * Innovations in end-user software push the limits of hardware, from productivity applications to games.

The demand for computing power is visible not only in our own lives on a micro-scale, but on the macro-scale as well. Machine-learning delays the GPU supply chain. Computing is a larger and larger fraction of a nation's entire electricity budget, with individual computing applications such as blockchains consuming more electricity than the entire demand of some small countries.

To meet the roaring demand for computation, two essential approaches have been code optimization and parallelization: using more processors to compute a larger task in the same time. Though optimization is essential, it is largely carried out through the development of optimizing compilers, a task that we defer to specialized texts on optimization due to the large number of low-level details required for a proper treatment. Parallelism, on the other hand, is amenable to a high-level analysis which is within the scope of this book.

We analyze the running time of programs in the following stages:

- * We determine the cost of running programs sequentially
- * We determine the cost of running programs with fork-join parallelism
- * We determine the cost of running programs with data parallelism

The performance of each execution mode is worth understanding. Sequential execution remains a common reality, and there exist algorithms which see no benefit from parallel execution. Fork-join parallelism is the most straightforward version of parallelism in the most straightforward to add to a functional programming language, and appears commonly in recursive algorithms. Data parallelism is used everywhere from vector processing in traditional CPUs to the GPUs which employ data parallelism at massive scale to address the world's largest computing problems.

The tool we use to achieve all this is called *cost semantics*, a version of operational semantics extended to track the cost of a computation, in number of execution steps. The following sections introduce cost semantics on a simple language, then extend it to parallel languages.

Sequential Cost Semantics

We first define a simple language used to explore cost semantics, then define the cost semantics, which are a form of big-step operational semantics.

Syntax

We introduce the syntax for a core language with support for numbers, pairs, and for some added complexity, first-class functions.

$$e \leftarrow x \mid n \mid (e, e) \mid \text{fun } f(x) \text{ is } e \mid e \text{ op } e \mid \text{fst}(e) \mid \text{snd}(e) \mid e(e)$$

As usual, x stands for a variable, n stands for a literal number, (e, e) stands for a pair, and $e \text{ op } e$ stands for an operator applied to its operands. To extract the elements of a pair, we use the expressions $\text{fst}(e)$ and $\text{snd}(e)$, which respectively return the first and second elements of the pair. Function application is written $e(e)$ because the function itself may be computed as the result of a non-trivial expression. The value $\text{fun } f(x) \text{ is } e$ stands for an anonymous function, a function that is a value. Specifically, it stands for the function f defined recursively by $f(x)=e$.

We assume a type system supporting the following types:

$$t \leftarrow \text{num} \mid (t, t) \mid t \rightarrow t$$

which correspond to numbers, pairs, and functions, respectively. We do not explicitly develop typing rules for this language because it uses standard rules presented elsewhere in this book.

Semantics

We are now ready to formally define the cost semantics. At its core, a cost semantics is an operational semantics that does some extra bookkeeping on the side to track the number of steps a program took to run.

We define the sequential cost semantics of our language as a single judgement, the judgement $E \vdash e \hookrightarrow v \in n$, where E is the environment in which program is evaluated to its value v, and n is the number of steps required for execution. We do not define a separate value judgement $E \vdash e \text{ value}$. In our slight abuse of notation, the symbol \in should *not* be interpreted as set membership, simply the English word “in”. The judgement is pronounced “In context E, expression e evaluates to v in n [steps]”. The mnemonic \in simply leaves off the word “steps”.

As is typical in a big-step semantics, we define rules for each construct of the language. We begin with the rules for variables and values. We assume that every operation, even returning a

value, consumes non-zero computing resources. For simplicity, we treat all operations has having the same unit cost 1. For practical implementations of analysis tools, separate costs could be provided for every operation.

Rule SeqVar

*

$$E \vdash x \hookleftarrow v \in 1$$

(where $E(x) = v$)

¶

Rule SeqNum

*

$$E \vdash n \hookleftarrow n \in 1$$

¶

Rule SeqFun

*

$$E \vdash (\text{fun } f(x) \text{ is } e) \hookrightarrow (\text{fun } f(x) \text{ is } e) \in 1$$

The next rule determines the cost of computing a pair. Pairs can be values, specifically if both their elements are values, however, pairs can also contain arbitrarily complex expressions, so we recursively evaluate their elements. To compute the overall cost of any compound expression, including pairs, we sum the cost of the premises evaluated and add a unit cost 1 to represent the added cost from the overall operation.

Rule SeqPair

$$E \vdash e1 \hookrightarrow v1 \in n1$$

$$E \vdash e2 \hookrightarrow v2 \in n2$$

$$E \vdash (e1, e2) \hookrightarrow (v1, v2) \in 1 + n1 + n2$$

The rule for evaluating operations has the same basic structure.

Rule SeqOp

$$E \vdash e1 \hookrightarrow v1 \in n1$$

$$E \vdash e2 \hookrightarrow v2 \in n2$$

$$E \vdash e1 \text{ op } e2 \hookrightarrow v1 \text{ op } v2 \in 1+n2+n2$$

The projection expressions $fst(e)$ and $snd(e)$ also recursively evaluate premises, but each one requires only a single premise:

Rule SeqFst

$$E \vdash e \hookrightarrow (v1, v2) \in n$$

$$E \vdash e \hookrightarrow v1 \in 1 + n$$

Rule SeqSnd

$$E \vdash e \hookrightarrow (v1, v2) \in n$$

$$E \vdash e \hookrightarrow v2 \in 1 + n$$

Function calls are the only rule that adds the cost of three operations:

Rule SeqCall

$$E \vdash e_1 \hookrightarrow (\text{fun } f(x) \text{ is } e) \in n_1$$

$$E \vdash e_2 \hookrightarrow v_2 \in n_2$$

$$(x \mapsto v_2) \vdash e \hookrightarrow v_3 \in n_3$$

$$E \vdash e_1(e_2) \hookrightarrow v_3 \in 1 + n_1 + n_2 + n_3$$

This completes the definition of the sequential cost semantics. To convince ourselves that our definition is correct, we could prove a theorem showing that the number of steps is computed correctly. We could prove this theorem in an exact fashion by appealing solely to cost semantics, or in an indirect fashion by comparison to a small-step semantics. We present theorem statements for both approaches.

Theorem: Let D be the formal derivation of judgement $E \vdash e \hookrightarrow v \in n$, then the number of nodes (rule applications) in derivation D is exactly n.

Proof: By induction on D. Exercise.

Theorem: Let $E \vdash e \mapsto_n e'$ denote that in environment E, program e steps to program e' in exactly n steps of small-step semantics. The

theorem states that whenever $E \vdash e \hookrightarrow v \in m$ holds, then $E \vdash e \mapsto_n e'$ holds for some $n \leq m$.

Proof: By induction on the derivation of $E \vdash e \hookrightarrow v \in m$. The exact number of steps may be lesser in small-step semantics because every value within an expression is counted as an operation by the big-step semantics. Exercise.

Fork-Join Cost Semantics

We now generalize the cost semantics to account for running programs in parallel. The model we consider is a form of *fork-join* parallelism, where independent subexpressions are run at the same time, but if an expression depends on multiple subexpressions, it must wait for them all to finish. Our analysis is intentionally idealistic: we assume that we have as many processors as we could ever use, so that some processor always remains idle. We assume that every opportunity for parallelism is taken, and that there is no overhead from forking an expression into parallel execution. Though idealistic, this semantics is appropriate for tasks such as analyzing the asymptotic complexity of parallel algorithms. Because all but the simplest parallel algorithms have some sequential dependencies within them, execution will take longer than constant time, even with arbitrarily many processors available.

When analyzing the cost of a parallel program, we care both about the total number of execution steps performed and the total time that would elapse when executing those steps in parallel. The total number of execution steps is the same as the cost of sequential execution, thus comparing these two metrics can assess the extent of parallel speedup. The total number of steps is also of interest because it could be used to estimate, e.g., the total cost of electricity for a large-scale computation. The total number of execution steps for an expression e is called the *work* of e , abbreviated $W(e)$, and the total time expended is called the *depth*^a of e , abbreviated $D(e)$.

To determine the work and depth of an expression, we follow these principles:

1. If e_1 and e_2 are performed in parallel, the work is $W(e_1) + W(e_2)$ and the depth is $\max(D(e_1), D(e_2))$
2. If e_1 and e_2 are performed one-after-the-other, the work is $W(e_1) + W(e_2)$ and the depth is $D(e_1) + D(e_2)$.

These short equations guide the design of the parallel cost semantics and, more importantly, have profound consequences for the performance of parallel code.

Our cost semantics is once again a big-step semantics, extended to consider both work and depth. The semantics consist of a single judgement $E \vdash e \hookrightarrow v \in (w,d)$, pronounced “in environment E, expression e evaluates to v in w work and d depth”. We present rules for each language construct, based on the principles of work and depth analysis.

The rules for values contain no surprises. Every value evaluates to itself in one step.

Rule WDVar

*

$$E \vdash x \hookrightarrow v \in (1,1)$$

(where $E(x) = v$)

Rule WDNum

*

$$E \vdash n \hookrightarrow n \in (1,1)$$

Rule WDFun

*

$$E \vdash (\text{fun } f(x) \text{ is } e) \hookrightarrow (\text{fun } f(x) \text{ is } e) \in (1,1)$$

To determine the cost of evaluating a pair, we observe that both elements can be evaluated in parallel, after which an extra step is required to construct the pair. For the first time, we see the work differ from the depth.

Rule WDPair

$$E \vdash e_1 \hookrightarrow v_1 \in (w_1, d_1)$$

$$E \vdash e_2 \hookrightarrow v_2 \in (w_2, d_2)$$

$$E \vdash (e_1, e_2) \hookrightarrow (v_1, v_2) \in (1+w_1+w_2, 1 + \max(d_1, d_2))$$

The rule for operations closely mirrors the rule for pairs.

Rule WDOp

$$E \vdash e_1 \hookrightarrow v_1 \in (w_1, d_1)$$

$$E \vdash e_2 \hookrightarrow v_2 \in (w_2, d_2)$$

$$E \vdash e_1 \text{ op } e_2 \hookrightarrow v_1 \text{ op } v_2 \in (1+w_1+w_2, 1 + \max(d_1, d_2))$$

The rules for projecting pairs of elements add a single step of both work and depth.

Rule WDFst

$$E \vdash e \hookrightarrow (v_1, v_2) \in (w, d)$$

$$E \vdash \text{fst}(e) \hookrightarrow v_1 \in (1+w, 1+d)$$

Rule WDSnd

$$E \vdash e \hookrightarrow (v_1, v_2) \in (w, d)$$

$$E \vdash \text{snd}(e) \hookrightarrow v_2 \in (1+w, 1+d)$$

The rule for function calls is the most subtle. The argument can be evaluated at the same time as the function is determined (i.e., as the function expression is evaluated to a function value).

However, execution of the function body cannot begin until the function and argument are both reduced to values. Thus, function calls introduce a sequential dependency which increases depth.

Rule WDCall

$E \vdash e_1 \hookrightarrow (\text{fun } f(x) \text{ is } e) \in (w_1, d_1)$

$E \vdash e_2 \hookrightarrow v_2 \in (w_2, d_2)$

$(x \mapsto v_2) \vdash e \hookrightarrow v_3 \in (w_3, d_3)$

$E \vdash e_1(e_2) \hookrightarrow v_3 \in (1+w_1+w_2+w_3, 1+\max(d_1, d_2) + d_3)$

This completes the definition of the cost semantics. In the process of developing these semantics, we observed that pairs are parallel and functions introduce sequential dependencies.

As before, we can either validate the cost semantics internally by analyzing their derivations, or we can compare them externally to a sequential semantics.

Theorem: Let D be the formal derivation of judgement $E \vdash e \hookrightarrow v \in (w, d)$, then the number of nodes (rule applications) in derivation D is exactly w and the height of D (number of rule applications on longest path from root to leaf) is exactly d.

Proof: By induction on D. Exercise.

Theorem: Let $E \vdash e \xrightarrow{n} e'$ denote that in environment E, program e steps to program e' in exactly n steps of small-step sequential

semantics. The theorem states that whenever $E \vdash e \hookrightarrow v \in (m, d)$ holds, then $E \vdash e \mapsto_n e'$ holds for some $n \leq m$.

Proof: By induction on the derivation of $E \vdash e \hookrightarrow v \in (m, d)$. The exact number of steps may be lesser in small-step semantics because every value within an expression is counted as an operation by the big-step semantics. Exercise.

Data-Parallel Cost Semantics

Once the fork-join semantics have been developed, data parallelism can easily be incorporated into the same language. This section is shorter than the others, but we include it to emphasize that the style of parallelism used for large-scale computation can be analyzed using the approach of cost semantics. We model data-parallel programs by extending the grammar of expressions

$$e \leftarrow \dots \mid [e, \dots, e] \mid e[e] \mid \text{map}(e, e)$$

where $[e_1, \dots, e_n]$ creates an array of n elements, each defined by the corresponding e_i , where $e[e]$ looks up an element of an array, and where $\text{map}(e, e)$ applies a transformation uniformly to every element of an array, producing a new array of the same length.

For full data-parallel programs, one might desire additional operations, which could be added seamlessly to our framework.

The judgement remains $E \vdash e \hookrightarrow v \in (w, d)$, as in the previous section. We define rules for each added construct.

WDArray

$$E \vdash e_1 \hookrightarrow v_1 \in (w_1, d_1)$$

...

$$E \vdash e_n \hookrightarrow v_n \in (w_n, d_n)$$

$$E \vdash [e_1, \dots, e_n] \hookrightarrow [v_1, \dots, v_n] \in (1 + w_1 + \dots + w_n, 1 + \max(d_1, \dots, d_n))$$

Our depth analysis for arrays is so optimistic as to be potentially controversial. We assume that an arbitrarily large amount of work can be distributed across processors in constant time, which accurately captures the dependency structure of a program but is likely inaccurate in a real implementation. When a more practical analysis is desired, alternative depth overheads (e.g., logarithmic) can be considered. This same commentary applies to the remaining operations as well.

WDIndex

$E \vdash e_1 \hookrightarrow [v_1, \dots, v_n] \in (w_1, d_1)$

$E \vdash e_2 \hookrightarrow i \in (w_2, d_2)$

$E \vdash e_1[e_2] \hookrightarrow v_i \in (1 + w_1 + w_2, 1 + \max(d_1, d_2))$

WDMAP

$E \vdash e_1 \hookrightarrow (\text{fun } f(x) \text{ is } e) \in (w, d)$

$E \vdash e_2 \hookrightarrow [v_1, \dots, v_n] \in (w', d')$

$(x \mapsto v_i) \vdash e \hookrightarrow v'_i \in (w_i, d_i) \text{ (for all } i \text{ in } [1, n])$

$E \vdash \text{map}(e_1, e_2) \hookrightarrow [v'_1, \dots, v'_n] \in (W, D)$

(where $W = 1 + w + w' + w_1 + \dots + w_n$, and $D = 1 + \max(d, d') + \max(d_1, \dots, d_n)$)

In $\text{map}(e_1, e_2)$, the operation e_1 is a function which gets called on each element of e_2 in parallel.

This completes the development of the data-parallel semantics, which can be proven to satisfy the same theorems as the fork-join semantics.

Related Work

This chapter is based on the theory of cost semantics, which has been assessed both numerically and using a call graph structure. Cost semantics have been used as the foundation of automated tool for analyzing the complexity of algorithms. Parallel implementations of many functional programming languages have been developed, following the principles discussed here.

Classroom Activities

- * Makes jokes about depth and about “Pairallelism” to earn the permanent ire of your students
- * Draw graphs to convey the dependency structure of a program
- * Write the semantics without the cost annotations, and interactively derive the annotations with the students
- * Consider demonstrating a parallel implementation of a functional program.

Exercises

- * Prove the theorems which were left as exercises.
- * Define the fork-join cost semantics rules of the following language features:
 - * Booleans, including conditionals
 - * Linked lists
 - * ADTs, such as binary trees
- * Implement the following, then analyze their (asymptotic) work and depth complexity. In some cases, the work and depth may be the same as each other.
 - * A naive (e.g., non-tail-recursive) implementation of the Fibonacci numbers.
 - * The mergesort sorting algorithm on linked lists.
 - * Insertion into a binary tree
- * Run a functional program using a parallel implementation and assess for yourself the amount of efficiency gained through parallelism.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor

+ cover image

bookish.press/book/chapter12

Chapter 21

More Instructor Materials

by Rose Bohrer + author (showing book authors)

Edit your chapter's title, authors, and cover image above. You can also change the ID of the chapter, which is the text that appears in the chapter URL. Write your chapter text below, formatting options in the toolbar above to add headers, lists, tables, comments, citations, footnotes, and more. It's okay to try things, you can always undo. Saves are automatic.

This chapter consists of assorted notes and instructions directed specifically to instructors, which are likely not of much interest to other readers.

Mid-Term Reflections

This note is a suggestion on how to run in-class mid-term reflections with students. This activity is designed for smaller class sizes, and may require modification for large lectures.

Start by sharing the following goals with the students

1. Give feedback how the course is going so far
2. Reflect on what the student has done so far
3. Plan what both the instructor and student should do in the second half

Course Feedback

In this section, collect feedback on what the instructor has done in the first half and what the students would like you to do in the second half.

Write the following prompts, e.g., on a board:

- * Stop Doing
- * Start Doing
- * Keep Doing

And ask for responses for each category, things you should stop doing, start doing, and keep doing as the instructor. Depending on the classroom layout and dynamics, students may give you verbal answers directly, or you might give them an opportunity to discuss amongst themselves and write responses directly.

This next set of prompts can either be used as an alternative to the previous set or used in parallel, in order to give students another

way to phrase their feedback. These prompts are usually not used sequentially, as they may duplicate each other's responses.

Make two columns on the board labeled “Issue” and “Solution.”

Ask students to provide an issue and its proposed solution. If the student is unsure of a solution, you can leave the spot blank in case someone else proposed a solution.

For both prompt sets, write +1 on remarks when students express agreement with them. Save all of the responses.

Self-Reflection

Before this in-class group reflection activity, it is recommended to assign every student a written self-reflection task, where they assess their current progress on course learning objectives, their interests, and their plans for the remainder of the course.

Encourage them to bring the results of this reflection to class for use in the group activity, if they so wish.

Encourage students to condense their self-reflection into:

- 1.** One thing they are proud of
- 2.** One thing they want to focus on in the remainder of the course

and then encourage them to share these responses with their classmates if they feel comfortable doing so.

If the reflection activity occurs immediately before a break, encourage students to expand #2 into a brief “letter to self” reminding them of their own plans when they return from the break.

Running User Studies in Class

This section provides advice on conducting user studies designed by students as part of their classwork. The exact method will depend on the size of the class. For small classes, it is often possible for every student to participate in the studies of every other student. For medium-size classes, students may be divided into groups which share the same room. For very large classes, it may be more practical to coordinate virtual studies over video call.

The studies are performed during class time. Set up a series of physical stations (e.g. desks) where an experimenter can sit while their subjects rotate through. The remainder of the setup is dependent to the number of stations (as an example here, 4).

Divide the students into batches whose size equals the number of stations. Divide the duration of the class period by the number of batches. This is length of time each student has to perform their experiments. If a batch has less than 20 minutes of experiment time, it is recommended to increase the number of stations or consider alternative methods. Even a 20-minute period is short, providing enough time, e.g., for a short 5-minute experiment to be repeated on 4 subjects.

Using a spreadsheet or a script, construct a schedule of arrival times for each student. Enough students should arrive at the beginning of the period to make use of each station and provide subjects for each station, but each student is free to depart once their batch has completed running experiments and serving as subjects for others. If students are working together in teams, extra flexibility can be provided, where the members of a team conduct the same study over a longer period of time, switching out with each other if needed.

Editors

These authors can edit this chapter. Add an email here to provide chapter-level edit permissions.

email

+ editor

rose.bohrer.cs@gmail.com book editor