EPTCS 405

Proceedings of the

Thirteenth Workshop on Trends in Functional Programming in Education

South Orange, New Jersey, USA, 9th January 2024

Edited by: Stephen Chang

Published: 10th July 2024
DOI: 10.4204/EPTCS.405
ISSN: 2075-2180
Open Publishing Association

Table of Contents

Table of Contents i

Preface ii
Stephen Chang

Invited Keynote: Essentials of Compilation: An Incremental Approach in Racket/Python iv
Jeremy Siek

Teaching Type Systems Implementation with Stella, an Extensible Statically Typed Programming Language 1
Abdelrahman Abounegm, Nikolai Kudasov and Alexey Stepanov

Functional Programming in Learning Electromagnetic Theory 20
Scott N. Walck

Finite-State Automaton To/From Regular Expression Visualization 36
Marco T. Morazán and Tijana Minić

Programming Language Case Studies Can Be Deep 56
Rose Bohrer

Preface

This volume of the Electronic Proceedings in Theoretical Computer Science (EPTCS) contains revised selected papers that were initially presented at the 13th International Workshop on Trends in Functional Programming in Education (TFPIE 2024). This workshop was held at Seton Hall University in South Orange, NJ, USA on January 9, 2024. It was co-located with the 25th International Symposium on Trends in Functional Programming (TFP 2024), which took place on January 10-12, 2024.

The goal of TFPIE is to gather researchers, teachers, and professionals that use, or are interested in the use of, functional programming in education. TFPIE aims to be a venue where novel ideas, classroom-tested ideas, and works-in-progress on the use of functional programming in education are discussed.

TFPIE workshops have previously been held in St Andrews, Scotland (2012), Provo, Utah, USA (2013), Soesterberg, The Netherlands (2014), Sophia-Antipolis, France (2015), College Park, MD, USA (2016), Canterbury, UK (2017), Gothenburg, Sweden (2018), Vancouver, Canada (2019), Krakow, Poland (2020), online due to COVID-19 (2021, 2022, with some talks from TFPIE 2022 also presented in person at Lambda Days in Krakow, Poland), and Boston, MA, USA (2023, back in-person).

The one-day workshop fosters a spirit of open discussion by having a two-phase, single-blind review process. First, the Programme Committee and Programme Chair lightly screen extended abstract submissions pre-workshop, to ensure that all presentations are within scope and are of interest to participants. After the workshop, presenters incorporate the feedback received pre-workshop and at the workshop, and are invited to submit revised versions of their articles for publication in the journal of Electronic Proceedings in Theoretical Computer Science (EPTCS).

This year we received seven total submissions pre-workshop. Five were presented in person, one online (due to travel issues), and one did not present (due to illness). At the workshop, we also had a keynote presented by Jeremy Siek of Indiana University titled “Essentials of Compilation: An Incremental Approach in Racket/Python.” We received post-workshop submissions from five out of seven pre-workshop submissions and accepted four. Each paper received at least two reviews in the pre-workshop round and three reviews in the post-workshop round. The final selections spanned a wide variety of interdisciplinary topics including type systems, programming language education, physics, and theory of computation.

All of this was only possible thanks to the hard work of the authors, the insightful contributions of the Program Committee members, and the thoughtful advice from the TFP/TFPIE steering committee. We also thank Epic Games and Seton Hall University for their financial support. Finally, we acknowledge Jason Hemann, the TFP 2024 chair, for organizing all the events. We thank everyone profusely for making this year’s symposium a success.

Program Committee

John Clements	Cal Poly, USA
Youyou Cong	Tokyo Institute of Technology, Japan
Christoph Lüth	University of Bremen, Germany
Simão Melo de Sousa	University of Algarve, Portugal
Vincent St-Amour	Northwestern University, USA

Stephen Chang (Ed.): Trends in Functional
Programming in Education (TFPIE)
EPTCS 405, 2024, pp. ii–iii, doi:10.4204/EPTCS.405.0

© Stephen Chang
This work is licensed under the
Creative Commons Attribution License.

Stephen Chang
Program Committee Chair and General Chair
University of Massachusetts Boston, USA
May 2024

Invited Keynote: Essentials of Compilation: An Incremental Approach in Racket/Python

Jeremy Siek
Indiana University
Bloomington, IN, USA
School of Informatics and Computing
jsiek@indiana.edu

This talk is an introduction to the joys of teaching and learning about compilers using the incremental approach. The talk provides a sneak-preview of a compiler course based on the new textbook from MIT Press, Essentials of Compilation: An Incremental Approach in Racket/Python. The course takes students on a journey through constructing their own compiler for a small but powerful language. The standard approach to describing and teaching compilers is to proceed one pass at a time, from the front to the back of the compiler. Unfortunately, that approach obfuscates how language features motivate design choices in a compiler. In this course we instead take an incremental approach in which we build a complete compiler every two weeks, starting with a small input language that includes only arithmetic and variables. We add new language features in subsequent iterations, extending the compiler as necessary. Students get immediate positive feedback as they see their compiler passing test cases and then learn important lessons regarding software engineering as they grow and refactor their compiler throughout the semester.

Stephen Chang (Ed.): Trends in Functional
Programming in Education (TFPIE)
EPTCS 405, 2024, pp. iv–iv, doi:10.4204/EPTCS.405.0.1

© Jeremy Siek
This work is licensed under the
Creative Commons Attribution License.

Teaching Type Systems Implementation with STELLA, an Extensible Statically Typed Programming Language

Abdelrahman Abounegm   Nikolai Kudasov   Alexey Stepanov
Lab of Programming Languages and Compilers
Innopolis University
Innopolis, Tatarstan Republic, Russia
a.abounegm@innopolis.university   n.kudasov@innopolis.ru   a.stepanov@innopolis.ru

We report on a half-semester course focused around implementation of type systems in programming languages. The course assumes basics of classical compiler construction, in particular, the abstract syntax representation, the Visitor pattern, and parsing. The course is built around a language STELLA with a minimalistic core and a set of small extensions, covering algebraic data types, references, exceptions, exhaustive pattern matching, subtyping, recursive types, universal polymorphism, and type reconstruction. Optionally, an implementation of an interpreter and a compiler is offered to the students. To facilitate fast development and variety of implementation languages we rely on the BNF Converter tool and provide templates for the students in multiple languages. Finally, we report some results of teaching based on students' achievements.

1 Introduction

Type systems constitute an important part of most modern programming languages, from Java and C++ to Python and TypeScript to Scala and Haskell, to mention a few. When using static types, programmers devote a significant part of their interaction with the compiler or a static analysis tool working through the type errors. Type-Driven Development (TyDD) [9] goes even further and suggests writing the types first and then follow the types to produce an implementation. Among popular modern programming languages, expressive type systems are becoming more widespread. Two recent examples include Rust [23], a systems programming language with ownership types, and Differentiable Swift [36], a dialect of Swift with differentiable types. Such languages incorporate quite elaborate type system features that require solid understanding of the basics to be used efficiently.

Type systems are not only used in compilers, but also in proof assistants such as Agda, Coq, and Isabelle/HOL. These usually rely on dependent type systems, such as Martin-Löf Type Theory [22] or Calculus of Constructions [10]. Using proof assistants for mathematics or program verification is somewhat similar in practice to TyDD in programming languages: in both cases the user of the system spends a significant amount of time dealing with type errors.

Although linters and good error messages help with type errors, it is important for the users to understand and be able to follow typing rules to productively resolve the issues and utilize type checker as a helping tool, instead of seeing it as a barrier. Although courses in statically typed programming languages help teach students some of the mechanics behind type checking, we believe a first-hand experience in the implementation of a type system deepens that understanding while also allowing students to apply some of those types in the implementation. Unfortunately, existing compilers construction courses appear to either provide an overview of the compiler pipeline, resort to a simplified semantic analysis phase, focus on the code generation phase, or work with a low-level intermediate representation such as LLVM IR or Java bytecode.

Stephen Chang (Ed.): Trends in Functional
Programming in Education (TFPIE)
EPTCS 405, 2024, pp. 1–19, doi:10.4204/EPTCS.405.1

© A. Abounegm, N. Kudasov & A. Stepanov
This work is licensed under the
Creative Commons Attribution License.

On the other hand, courses that focus on type systems often offer implementation of a variant of typed lambda calculus, which appears to disengage some students who are used to Java-style or Python-style syntax of programming languages and who do not have sufficient experience with functional programming. In particular, in a previous iteration of our course, many students found it quite difficult and unintuitive to program or even read programs in lambda calculus, whereas rewriting a lambda expression in Python using explicit function definitions helped them navigate the operational semantics of lambda calculus.

While lambda calculus is too raw, any modern programming language is too complex. Indeed, even the core of most modern languages is too complex to give as an implementation exercise for students. The Haskell programming language, and in particular its implementation in the Glasgow Haskell Compiler (GHC), comes close to a language with a small core: it employs a system of language extensions that allows to enrich the language per module. In this work, we are inspired by this approach and design a language with a very small fixed core that is easy to implement for students, and provide various extensions for further developments.

To be able to focus on the type checking, it is not enough to assume that students have already mastered the lexical and syntactical analyzers. Implementing those with proper types for the abstract syntax tree from scratch still requires significant time. Luckily, at least for simple languages, there is a good selection of tools that can do that automatically, such as XText [12] and BNF Converter [13]. In our course, we rely heavily on BNF Converter to offer templates in multiple languages and encourage diversity of implementations in student submissions.

1.1 Related Work

The idea of designing a programming language specifically for teaching compiler construction is not new. One of the first such languages was `MINIPASCAL`, a simplified version of Pascal programming language, introduced by Appelbe [5].

More recently, Berezun and Boulytchev [6] reported on the use of the programming language $\lambda a \mathscr{M} a$[1], developed by JetBrains Research for educational purposes, in their introductory course on compilers. They asked students to implement a compiler of $\lambda a \mathscr{M} a$ in itself, a procedural language with first-class functions that is untyped (meaning no *static* type checking is performed). It appears that the choice to make the language untyped was to afford simpler implementations. Thus, Berezun and Boulytchev's course focuses on parsing, basic semantic analysis, and code generation. In contrast, our course focuses on type systems and puts semantics first.

Aiken [1] introduced the Cool language, which is largely inspired by Java and features a static type system and automatic memory management. Cool's type system is nominal, without support for generics, anonymous objects, or higher-order functions, and thus does not present significant challenges in the implementation of type checking compared to the structural type system featured in our course.

Although basic typechecking can be covered in regular compilers courses, some courses still specialize specifically on type systems and type checking. Ortin, Zapico, and Cueva [29] teach design patterns helpful for implementing a type checker in object-oriented languages.

Lübke, Fuger, Bahnsen, Billerbeck, and Schupp [19] show how to automate exams and assignments for functional programming that include proofs traditionally performed on paper. With STELLA's system of extensions, we have the basis for similar automation for our assignments, but we keep advanced automation for future work.

[1] `https://github.com/PLTools/Lama`

1.2 Contribution

In this paper, we report on a half-semester course on the implementation of type systems. More specifically:

1. In Section 2, we overview the STELLA language, consisting of a minimalistic purely functional core and a set of extensions, matching multiple topics covered by Pierce in his book [31].
2. In Section 3, we specify our approach to template solutions, allowing students to get started with implementations in any of a multitude of languages, including C++, Java, Kotlin, OCaml, TypeScript, Swift, Rust, Go, and Python.
3. In Section 4, we outline the course structure and identify both practical and theoretical tasks for the students. Here, we also evaluate the course based on students' performance.

2 Overview of the STELLA Language

We have designed and implemented the STELLA[2] language specifically for the purposes of teaching type systems implementation following Pierce's book [31]. In this section, we highlight the main design decisions behind the STELLA language and outline its main features.

STELLA is a statically-typed expression-based language. It consists of a minimalistic core language and a set of extensions. Making language extensions explicit is advantageous in several ways:

1. Students can quickly learn the core, explicitly see a list of features used in every example program, and know where to look for corresponding documentation;
2. Test programs explicitly specify the language extensions used, making it easy to understand which features (or combinations of features) are supported by a student's implementation;
3. Students can ignore extension pragmas, implicitly supporting any given set of extensions, simplifying implementations;
4. More extensions can be added to STELLA in the future.

The canonical implementation of Stella supports all extensions. Templates provided to the students also support all extensions, meaning that the abstract syntax types are provided for the full language. Although this slightly complicates the types for the core language (e.g. students have to deal with a list of function arguments instead of a single argument), we find that in practice students do not have any major issues with this.

A program in STELLA always consists of a single file since a module system is out of the scope of the course as it adds unnecessary complexity while not adding much to the typechecking part of the compiler.

Below, we describe the core part of the language and the main extensions featured in the course.

2.1 STELLA Core

The core language of STELLA is essentially a simply-typed functional programming language, based on Programming Computable Functions (PCF) [32], except having a syntax inspired particularly by Rust [23], and reasonably understandable to many programmers. In fact, one of the main motivations

[2]**S**tatically **T**yped **E**xtensible **L**anguage for **L**earning **A**dvanced compiler construction (type systems)

```
1  // sample program in Stella Core
2  language core;
3
4  fn increment_twice(n : Nat) -> Nat {
5    return succ(succ(n))
6  }
7
8  fn main(n : Nat) -> Nat {
9    return increment_twice(succ(n))
10 }
```

Figure 1: Simple program in STELLA Core.

for STELLA was the fact that many students struggled with the *syntax* of λ-calculus but many examples were better understood (and students came up more easily with their own examples) when moving to a more familiar syntax of pseudocode or Python, Java, TypeScript.

STELLA Core also corresponds to simply-typed lambda calculus [31, §9] with two base types (natural numbers and booleans) [31, §11.1].

Importantly, STELLA Core supports first class functions. We justify support for higher-order functions from the start by asking students to only implement a typechecker, leaving the overview of possible implementations for a proper compiler for a functional language to a later part of the course. Indeed, while a compiler might need to deal explicitly with closures and renaming of bound variables, in a typechecker (without extensions like universal polymorphism or dependent types) handling of scopes is straightforward and is not affected by the presence of higher-order functions. At the same time, allowing higher-order functions allows us to immediately introduce more sophisticated test programs.

A sample program in STELLA Core is given in Fig. 1:

1. Line 1 is a comment line; all comments in STELLA start with `//`; there are no multiline comments in STELLA;

2. Line 2 specifies that we are using just STELLA Core (without any extensions); it is mandatory to specify the language in the first line of a Stella program;

3. Line 4 declares a function `increment_twice` with a single argument `n` of type `Nat` and return type `Nat`; in STELLA Core, all functions are single-argument functions;

4. In Line 5, `return ...;` is a mandatory part of every function; in STELLA Core each function can only return some expression; there are no assignments, operators, or other statements possible;

5. In Line 5, `succ(n)` is an expression meaning $n+1$ (the successor of n);

6. Line 8 declares a function `main` with a single argument `n` of type `Nat` and return type `Nat`; in a Stella program there must always be a `main` function declared at the top-level; the argument type and return type of `main` can be specified to be any valid types;

7. In Line 9, `increment_twice(succ(n))` is an expression meaning "call `increment_twice` with the argument `succ(n)`".

2.2 Extensions

STELLA supports a number of language extensions, which can be enabled with an `extend with` pragma, containing a list of extension names separated by commas. Each extension may add new syntax, typing rules, or other capabilities to the language. Following Pierce [31], we separate *Simple Extensions* from some more advanced type system extensions. However, we also separate *Syntactic Sugar and Derived Forms* and *Base Types* into standalone categories.

In general, each extension is supposed to be small enough to be implemented either as a single exercise or at most as a standalone assignment. For example, each base type has its own extension. Some extensions have simple and generalized versions. For instance, `#tuples` generalizes `#pairs` by allowing arbitrary size and `#variants` generalizes `#sum-types`.

2.2.1 Syntactic Sugar and Derived Forms

This category contains language extensions that *may* be implemented as derived forms, reducing them to other features of STELLA. However, this is not enforced and students are allowed to implement these extensions as standalone features. Some notable extensions in this category include `let`-bindings, nested function declarations, multiparameter functions, automatic currying, and type ascriptions.

A simple, but important extension is sequencing. Coupled with effectful expressions (such as mutable references), this enables imperative programming features.

2.2.2 Nested Pattern Matching

Some extensions (such as sum types or variants) naturally come equipped with pattern matching constructions. However, by default they do not allow nested patterns, to simplify implementation and first example. *Nested* pattern matching is available via the `#structural-patterns` extension. This extension allows nesting and combining patterns that are enabled by other extensions. Nested patterns complicate *pattern-match coverage checking* (also known as *exhaustiveness checking*) even just in presence of both tuples and variants and although it is a well-studied problem [18, 21, 34] we leave the implementation of the exhaustiveness checker out of the scope of our course and only ask for exhaustiveness checks without nested patterns.

2.2.3 Simple Types

The simple types correspond directly to Pierce's *Simple Extensions* [31, §11.2, §11.6–11.10, §11.12] and include the `Unit` type, pairs, tuples, records, sum types, variants, and (built-in) lists.

Importantly, all these types are *structural* rather than *nominal* [31, §19.3].

2.2.4 References

STELLA introduces references following Pierce [31, §13]. Without any ownership types, this does not introduce any complications in the typechecking.

2.2.5 Exceptions

Following Pierce [31, §14], we introduce support for different treatment of exceptions in STELLA:

```
language core;

extend with #exceptions, #exception-type-declaration;

exception type = Nat

fn fail(n : Nat) -> Bool {
  return throw(succ(0))
}

fn main(n : Nat) -> Bool {
  return try { fail(n) } with { false }
}
```

Figure 2: A STELLA program featuring exceptions.

1. First, simple (unrecoverable) errors [31, §14.1] are supported in the `panic!` expression, enabled through `#panic` extension;
2. To throw and catch exceptions carrying values [31, §14.3], the extension `#exceptions` is used. To specify the type of values in exceptions, the user has a choice:
   (a) One option allows user to fix the type of values carried by exceptions. This is achieved with the `#exception-type-declaration` extension and requires a specification of exception type:
   ```
   // use error codes for exceptions
   exception type = Nat
   // use a fixed variant type for exceptions
   exception type = <| error_code : Nat, good : Bool |>
   ```
   Fig. 2 provides an example of a complete STELLA program that features exceptions.
   (b) Another option is to use open variant type for exceptions (OCaml-style). This is achieved with the `#open-variant-exceptions` extension and allows adding variants to the exception as needed. The following two declarations are equivalent to the explicit variant type above, except, with open variant, it can also be extended with more variants later:
   ```
   exception variant error_code : Nat
   exception variant good : Bool
   ```

At the moment, STELLA does not support any annotations that would specify the possible exceptions thrown by a function.

2.2.6 Subtyping

STELLA supports structural subtyping, following Pierce [31, §15]. The subtyping mechanism is enabled with the `#structural-subtyping` extension and interacts with other enabled extensions, such as `#records` and `#variants`.

In Fig. 3 on line 10, the function `getX` is applied to a record of type `{x : Nat, y : Nat}`. Since this type is a subtype of `{x : Nat}`, it is accepted when structural subtyping is enabled. Similarly, in

```
language core;

extend with #records, #structural-subtyping;

fn getX(r : {x : Nat}) -> Nat {
  return r.x
}

fn main(n : Nat) -> Nat {
  return getX({x = n, y = n});
}
```

Figure 3: Sample STELLA program with structural subtyping and records.

Fig. 4 on line 17, the function `inc` is applied to a value of variant type `<| value : Nat |>`. Since this type is a subtype of `<| value : Nat, failure : Unit |>`, it is accepted when structural subtyping is enabled.

Subtyping for function types is enabled automatically and presents a challenge for students, since it requires them to properly understand the idea of covariant and contravariant subtyping. We find this also to be one of the most valuable learning points for the students, since even without typing, these concepts are important in programming.

Additional extensions introduce `Top` and `Bot` (short for "Bottom") types as well. The `Top` type is the supertype of all types, while the `Bot` type is the subtype of all types. The `Top` type is not very useful on its own, so is normally used in combination with the `#type-cast` extension, which adds the `<expression> cast as <type>` expression syntax. This operator performs downcasting of the expression to the specified type, which only works if the expression was already of a supertype of the specified type.

STELLA does not currently support intersection or union types [31, §15.7].

2.2.7 Universal Polymorphism

Universal types [31, §23] are implemented in Stella in the form of generic functions that accept type arguments. This is enabled by the `#universal-types` extension. To declare a generic function, one needs to add the `generic` keyword before `fn` and add a type parameter list after the function name. The type parameters are a comma-separated list of type variables enclosed in square brackets. When invoking a generic function, the type arguments must be provided in square brackets after the function name.

Fig. 5 shows a sample program with universal polymorphism:

1. In Line 5, we declare function `id` that is parametrized by type `T`;
2. In Line 10, we apply the function `id` to an argument, explicitly instantiating the type parameter to `Nat`.

Fig. 6 features a universally polymorphic anonymous function:

1. In Line 5, we declare the function `const` that returns a parametrically polymorphic function of type `forall Y. fn(Y) -> X`;
2. In Line 6, we construct an anonymous function of said type;

```
1  language core;
2
3  extend with #variants, #structural-subtyping;
4
5  fn inc(r : <| value : Nat, failure : Unit |>) -> Nat {
6    return match r {
7        <| value = n |> => succ(n)
8      | <| failure = _ |> => 0
9    }
10 }
11
12 fn just(n : Nat) -> <| value : Nat |> {
13   return <| value = n |>
14 }
15
16 fn main(n : Nat) -> Top {
17   return inc(just(n));
18 }
```

Figure 4: Sample STELLA program with structural subtyping and variants.

3. In Line 10, we first instantiate the type parameter of `const` to `Nat`, then pass `x` argument to get a parametrically polymorphic function of type `forall Y. fn(Y) -> Nat` as a result. Finally, we instantiate `Y` with `Bool` and pass `false` as an argument.

Currently, STELLA does not support any form of bounded quantification [31, §26], as a half semester course does not have enough room to explore this important topic in details. However, the topic is briefly mentioned in the lectures. In particular, the undecidability of type reconstruction in presence of bounded quantification [31, §28.5.5] is discussed.

The `#universal-types` extension enables unrestricted impredicative universal types as in System F [14, 33]. The following example demonstrates impredicativity through self-application of `f` to itself:

```
generic fn self_app[X](f : forall X . fn(X) -> X) -> forall X . fn(X) -> X {
  return f[forall X . fn(X) -> X](f)
}
```

In the future, we aim to support various kinds of universal polymorphism in STELLA, at least allowing for Hindley-Milner-style, predicative, and bounded quantification (parametric polymorphism with subtyping) [31, §26].

2.2.8 Recursive Types

Recursive types [31, §20] (specifically, *iso-recursive types*) are technically implemented in STELLA, but not used in the course. The reason for that is that most programming languages rely on nominal typing to provide recursion in types. Thus, in the course, we discuss recursive types in theoretical materials, but do not offer implementation of this part of STELLA.

```
1  language core;
2
3  extend with #universal-types;
4
5  generic fn id[T](x : T) -> T {
6    return x
7  }
8
9  fn main(x : Nat) -> Nat {
10   return id[Nat](x)
11 }
```

Figure 5: Sample STELLA program with universal polymorphism.

```
1  language core;
2
3  extend with #universal-types;
4
5  generic fn const[X](x : X) -> forall Y. fn(Y) -> X {
6    return generic [Y] fn(y : Y) { return x }
7  }
8
9  fn main(x : Nat) -> Nat {
10   return const[Nat](x)[Bool](false)
11 }
```

Figure 6: Sample STELLA program with a universally polymorphic anonymous function.

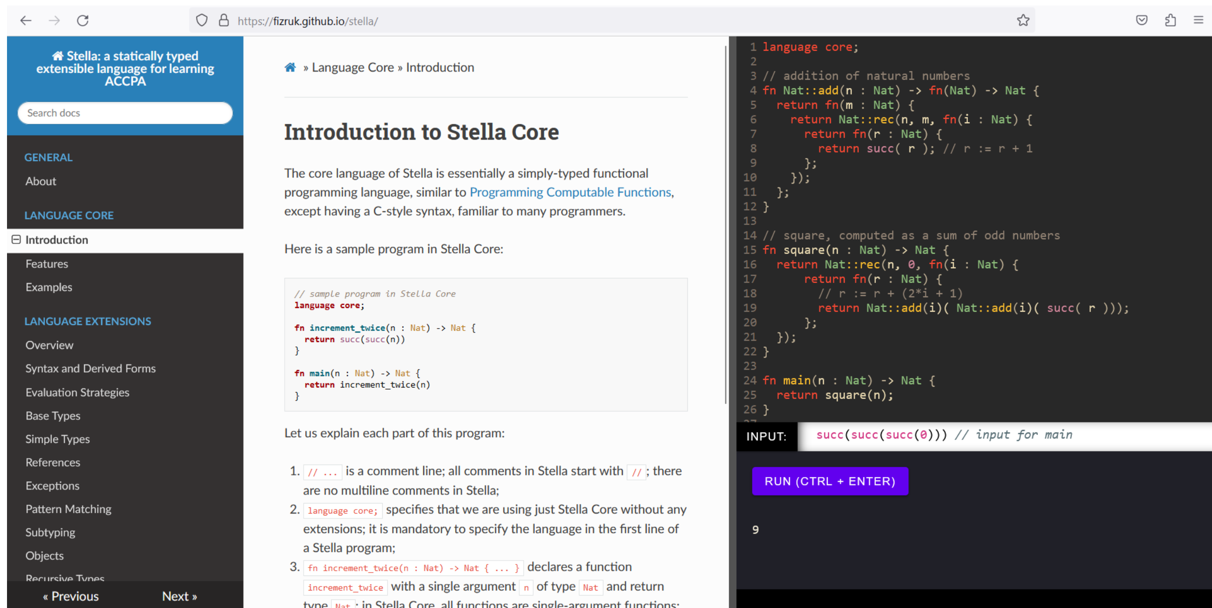


Figure 7: Stella documentation and playground.

2.3 Accessibility: Documentation and Interactive Playground

To provide the students with a convenient way to learn the language and its extensions, we have created a website with documentation and an interactive playground. The website is available at `https://fizruk.github.io/stella/`.

The documentation is written in Markdown and is compiled to HTML using MkDocs [3]. The browser version of the Stella interpreter is compiled using GHCJS [28] and Miso framework[4], which allow us to compile Haskell code to JavaScript and run it in the browser. The playground consists of a CodeMirror[5] code editor with custom syntax highlighting using Highlight.js[6]. The playground includes an input field which acts as the `main` function argument, and an output field which displays the result of the `main` function or the compilation errors (if any).

Additionally, a VS Code extension is available for Stella, which provides syntax highlighting and snippets for the language[7].

3 Implementation Templates and Typechecker Structure

By design, students are free to select any implementation language in the course. To facilitate with implementations in many languages and avoid eating up the time for parsing, abstract syntax implementation, and pretty-printing, we provide students with an EBNF grammar, suitable for the BNF Converter tool [13]. See Section 4 for the course setup, where we explain the prerequisites allowing us to have a fast setup and skip a major part of the frontend for the student implementation. The tool allows us to

[3] `https://www.mkdocs.org`
[4] `https://github.com/dmjio/miso`
[5] `https://codemirror.net/`
[6] `https://highlightjs.org/`
[7] `https://github.com/IU-ACCPA-2023/vscode-stella/`

generate a lot of structure in many languages, including Haskell, OCaml, Java, and C++. To reach even more languages, we have leveraged the Java ANTLR backend of BNFC to produce ANTLR grammar with Java code snippets. We then manually removed the Java code from the ANTLR grammar and added extra annotations to provide a language-agnostic ANTLR file[8]. Using ANTLR backends for other languages and writing down some definitions and conversions, we have been able to get decent templates for more languages, including Rust, Swift, and TypeScript.

3.1 A Zoo of Implementation Languages

For convenience of students, we have prepared project templates[9] in several programming languages. The templates allow to immediately proceed with the solution of the course objectives (implementation of typecheckers), without spending extra time on the implementation of lexer and parser, definition of the abstract syntax, or implementing a pretty-printer. The templates were prepared for the following languages: C++, Java, OCaml, Python, Rust, Go. During the course, at the request of the students, and with their help, the following languages were added to the list: Kotlin, Swift, TypeScript, Haskell.

In the course, each student selects an implementation language, creates a copy of a suitable template, completes the coding exercises, and submits the solution via GitHub Classroom (by pushing to the remote repository created for a particular assignment).

The templates were initially designed with the following structure, mostly provided by BNFC:

1. A grammar file (Labelled BNF grammar) describing the STELLA language;
2. Lexer and parser generated by the BNFC tool based on the grammar file;
3. The types (e.g. classes) for the abstract syntax of STELLA;
4. A generated skeleton for recursive traversal of the abstract syntax;
5. The main project file, which contains the functionality to read STELLA source and convert it to AST;
6. A `Makefile` to compile and run the project;
7. Some tests to verify the success of building and running the project.

At the start of the course, we found out that many students are not comfortable enough with `Make`, so we have accommodated for more IDE-friendly setups, relying on language-specific build systems, such as CMake for C++, Dune for OCaml, and Maven for Java.

The Kotlin template is based on the Java pattern, meaning that the parser is implemented using Java classes. However, the interpreter code is written in Kotlin and the build system has been replaced by Gradle.

The ANTLR grammar produced by BNFC for the Java backend was used as the basis to accommodate templates for more languages. It was manually rewritten into a language-agnostic form (removing Java code, adding labels for rules, and slightly modifying the structure of the grammar). We then leveraged ANTLR to generate the *context types* for concrete syntax tree and manually added types for abstract syntax together with corresponding conversion functions for each of the extra languages.

[8] see, for instance, `https://github.com/IU-ACCPA-2023/stella-implementation-in-swift/blob/main/Sources/stella-implementation-in-swift/Stella/stellaParser.g4`

[9] `https://github.com/IU-ACCPA-2023`

4 Course Structure

In this section, we discuss the organization of the course, its materials, the instruments used, and the results of the course.

The course consists of two lectures and two lab sessions each week for a total of 15 lectures and 15 labs over the course of eight weeks. There are 4 coding and 3 theoretical assignments in the course, as well as an optional oral exam at the end of the course. The first 6 weeks of the course cover type systems following Pierce [31, §8–23], and the last 2 weeks cover implementation details for lazy functional languages, following Peyton Jones [17, §1–5].

The course is focused on the study of type systems, their properties and implementation. For the theoretical part of the course, students are expected to learn about type safety, canonical representation, normalization, and be able to prove these properties for simple type systems. The practical part consists of implementing a type checker for STELLA with a subset of extensions. Since many of the students taking the course already have experience with statically typed languages like Java, C#, C++, Scala, Kotlin, Go, Dart, and TypeScript, an implicit additional objection of the course is to help students better understand and utilize type system features in those languages, as well as appreciate the diversity of type systems and practical choices made by designers of those systems.

4.1 Prerequisites

For the most part, the course emphasizes a detailed study of a specific component of compilers - the type checker. Our course immediately follows another half-semester introductory course on compiler construction, during which students developed the skills to construct their own compiler from the ground up. Thus, entering our course, students are expected to have a general understanding of the structure of a compiler and be especially comfortable dealing with the abstract syntax. Familiarity with compilers from textbooks by Wirth [37], Muchnick [27], and Appel [2, 3, 4] is assumed. Therefore, our course omits these aspects and focuses on more in-depth semantic analysis, type checking [31, §8–23], and runtime for lazy functional languages [17].

The course additionally assumes familiarity with some statically typed programming languages. In particular, we assume students to be able to write object-oriented code in C++ or Java, and be able to write functional code in Haskell, OCaml, Scala, or a similar language.

The course load is approximately 15 hours per week. This includes two lectures, two laboratories, and an estimated time for homework. In addition to this course, students are enrolled in two other courses, which are also organized in a block scheme.

4.2 Blocks

The course was organized into five blocks as the lecture topics and language functionality became more complex. The STELLA language evolves thoughout the course from the core language, which is a simple functional language, to a language with imperative objects [31, §18] or an ML-style language with universal polymorphism [31, §23]. The organization of blocks was as follows:

Block 1. Simple Types

The first block covers the terminology and fundamental concepts necessary to completely understand simply-typed lambda calculus and implement typechecking for STELLA Core, as well as some simple extensions from the *Syntactic Sugar and Derived Forms*, *Base Types*, and *Simple Types* categories.

The coding assignments in this block require students to implement:

1. A typechecker for STELLA Core, covering `Nat`, `Bool`, and function types.
2. A typechecker for simple types: `Unit`, pairs, and sum types.
3. For extra credit, students may implement a typechecker with support for some of additional extensions: variants, tuples, records, `let`-bindings, `letrec`-bindings, nested pattern matching (without exhaustiveness checks), type aliasing, and general recursion.

A theoretical assignment is issued, focusing mainly on sum types and pattern matching, making sure, students properly understand these concepts.

Block 2. Normalization and Recursive Types

The second block covers theoretical topics, discussing normalization properties of simply typed lambda calculus [31, §12] and talking about recursive types [31, §20]. No new assignments are issued in this block, since students have an active assignment from the previous block.

Block 3. Imperative Objects

In the third block, we focus on adding imperative features like references [31, §13] and exceptions [31, §14], and also discuss subtyping, aiming to arrive at a language that supports imperative objects [31, §18]. We also discuss Featherweight Java [16] and Welterweight Java [30]. The latter extends Featherweight Java with imperative, stateful features, as well as thread-based and lock-based concurrency. While STELLA does not support nominal types, students are still asked to complete theoretical excercises using Welterweight Java.

The coding assignment in this block requires students to implement:

1. A typechecker supporting sequencing, mutable references, unrecoverable errors, and records with subtyping.
2. For extra credit, students may implement a typechecker with support for some of the additional extensions: subtyping for variants, exceptions with a fixed (super)type, `Top` and `Bot` types, and exceptions with an open variant type.

Block 4. Type Reconstruction and Universal Types

The fourth block focuses on *type reconstruction* [31, §22] (aka *type inference*) and universal types [31, §23]. Additionally, we discuss Hindley-Milder type system [15, 26] and the type inference algorithm for it [11].

The coding assignment in this block requires students to implement type variables, universal types, and generic function declarations. The implementation of the type checking here is complicated since students have to take care with type variables bound by the `forall` quantifier in the polymorphic types.

Block 5. Runtime for Lazy Functional Languages

This last block is dedicated to the exploration of runtime for lazy functional languages. While this does not directly correspond to type checking, it helps students appreciate sum types and variants more, as they see how they can be efficiently implemented in a language like Haskell.

4.3 Assessment

Once a week at the start of the lecture, students complete a brief quiz consisting of simple theoretical questions (typically, multiple-choice or matching questions). The quiz is intended to be finished in the classroom within 10 minutes and serves mostly as a tool to facilitate attendance and wake students before the lecture.

To assess the students more comprehensively, three theoretical problem sets are prepared and assigned as homework for one week each. In the labs, students are working on their coding exercises and receive help from teaching assistants. Most assignments have an extra credit part.

Students' grades are computed based on completion of assignments and quizzes. In case of late submissions or to allow for an improvement of their grade, students are allowed to attend an oral exam to defend their implementation and answer some more theoretical questions for some additional credit.

4.3.1 Solutions for Assignment 1

After completing the first coding assignment, we provide students with a reference implementation in C++ and Java. In our course, these languages turned out to be most popular among the students, but presented particular implementation challenges. For instance, in C++, students struggled with the complex handling of raw pointers and segfaults, partially due to raw pointers in the generated code from BNFC. An additional motivation for sharing our solution with the students was that the first assignment is the basis for all subsequent assignments, and it is important that it has a good structure and the ability to be easily extended. This allowed the students to compare and refine their implementations or build entirely on ours. We find that without a reference implementation for the first assignment, many students would waste time rewriting the basic logic for each assignment, which is unacceptable given the course's time frame.

4.3.2 Collecting Student Submissions

For quizzes, we relied on the Moodle platform provided by Innopolis University and automatically graded using it. Solutions to theoretical problem sets have been also submitted through Moodle as PDF files, then graded manually.

For coding exercises, we considered using platforms such as Codeforces[10]. Unfortunately, such platforms usually deal with a single-module submissions, which cannot accommodate for the complex compiler project structure.

Consequently, we redirected our focus to GitHub Classroom[11] due to its advantages, including the capability to create repositories in various programming languages as templates for students to clone and commence their projects, as well as to monitor student submissions by tracking commits. Although we chose GitHub Classroom, it turned out not ideal for the following reasons:

1. It is difficult to update the assignment template (e.g. fix problems, add tests).
2. It is impossible to offer templates in different languages.
3. Consecutive assignments have separate repositories, it is impossible to submit an individual commit as a solution to a particular assignment. To start a new assignment, students had to clone the template each time and manually apply their changes from the previous assignment.

[10] `https://codeforces.com/`
[11] `https://classroom.github.com/`

In the end, taking into account the aforementioned issues, GitHub Classroom has allowed us to keep track of students' work throughout the course: tracking their submission times, work completed, and assessments.

4.4 Results

The course was conducted for two groups with a total of 49 (active) students. A final exam, which was optional and offered to all, was taken by approximately 20% of the class. To provide students with the ability to obtain additional guidance, make inquiries, and receive notifications, we provided a Telegram chat room in addition to the Moodle and GitHub Classroom platforms previously mentioned. Students completed their projects using various programming languages.

Students selected different languages for their implementation. The distribution of submissions by language is presented in Table 1. Students have used C++ and Java extensively in the previous courses, so these were the natural choices for most students. Kotlin was chosen by many students even when they did not know the language before, due to simpler code structure. In particular, while the Java template featured the Visitor pattern which many of the students struggled with (despite being exposed to it in other courses), in Kotlin students were able to use exhaustive pattern matching through `when`-expressions, which they found very convenient when dealing with the nodes of the abstract syntax tree.

Students had some freedom in selecting a subset of extensions to implement in their submissions. In Table 2, we show for each feature, how many students have implemented it fully (passing all tests) or partially. *Universal Types* extension was not implemented correctly by many students due to the problem of name captures, which is non-trivial to implement and also does not occur in small test programs that students used for debugging. Many students who attempted to implement *Stuctural Patterns* struggled with handling contexts properly. Often the reason was due to incorrect handling of mutable state in an implementation of the Visitor pattern. When students approached the implementation with a functional style, they were less likely to make mistakes.

At the beginning of the course, we have encountered a major problem when students were not able to work with the `Makefile`-based templates that we have prepared originally. Setting up the environment and getting the project to a point where they could begin writing code took one week, which has affected the schedule of the course negatively, and forced us to skip some assignments. Having learned from this experience, we plan to enhance IDE-friendly templates for the future installation of the course.

Table 3 presents the distribution of the final grades for the course according to different programming languages. With a relatively low number of students, it is hard to draw any definitive conclusions from this data. It is perhaps not so surprising to see many good grades for popular languages like Java, Kotlin, and C++, since these languages had most support from the course team, templates, and students were free to communicate and help each other in case of technical difficulties. On the other hand, it is interesting to note that students who selected languages that initially lacked templates for implementation (Swift, Haskell, TypeScript) also achieved good grades. The students who used these languages are relatively autonomous and proficient, but we also believe that these languages possess features (such as pattern matching) that noticeably simplify development of certain parts of a typechecker, which might have helped these students as well.

Programming Language	Submissions
C++	17
Java	12
Kotlin	12
Python	3
JavaScript	2
Haskell	1
Swift	1
TypeScript	1

Table 1: Language distribution.

Feature	Full	Partial	Full, %	Partial, %
Core	49	0	100%	0%
Records	46	0	94%	0%
Pairs	45	0	92%	0%
Unit type	44	0	90%	0%
Sequencing	43	0	88%	0%
References	42	0	86%	0%
Sum types	39	0	80%	0%
Errors	39	0	80%	0%
Subtyping for records	39	0	80%	0%
Tuples	38	3	78%	6%
Universal types	17	21	35%	43%
Top and Bot types	14	8	29%	16%
Records	12	5	24%	10%
Exceptions with a fixed type	11	0	22%	0%
Exceptions with an open variant type	10	0	20%	0%
LetRec-binding	9	0	18%	0%
Subtyping for variants	9	0	18%	0%
Variants	7	0	14%	0%
Let-binding	7	2	14%	4%
Type aliases	6	1	12%	2%
Structural patterns	3	21	6%	43%
General recursion	3	20	6%	41%

Table 2: Implemented features in numbers.

Language	A	A, %	B	B, %	C	C, %	D	D, %
C++	13	76%	1	6%	3	18%	0	0%
Java	12	75%	2	17%	0	0%	1	8%
Kotlin	8	67%	4	33%	0	0%	0	0%
Python	1	33%	1	33%	0	0%	1	33%
JavaScript	1	50%	1	50%	0	0%	0	0%
Haskell	0	0%	1	100%	0	0%	0	0%
Swift	1	100%	0	0%	0	0%	0	0%
TypeScript	0	0%	1	100%	0	0%	0	0%

Table 3: Language to grades.

5 Conclusion and Future Work

We have presented a half-semester course that focuses on the study and implementation of type systems, supported by a special programming language STELLA designed to facilitate students' learning process. The language is intended to accompany a well established textbook by Pierce [31], and our preliminary experience shows that students that are used to C-like syntax absorb the textbook material better when implementing STELLA, at least compared to implementing raw typed lambda calculi.

There are still many ways to improve STELLA to better accommodate the needs of the educational process. Nominal type systems prevail in modern programming languages, so STELLA should support them to enable a more fruitful discussion and comparison with structural types. Bounded universal quantification is currently not supported in STELLA, but we plan to add it, since it allows for a more direct experience with type systems that incorporate both parametric polymorphism and subtyping, and is used a lot in languages like Java, Scala, Kotlin, C#, OCaml. Annotating functions with the types of possible exceptions is also currently not supported and provides for a relatively straightforward but useful language extension. Row polymorphism [35] is a relatively rare but important concept, that is currently not explored by STELLA.

Although we focus mostly on type systems, we are also interested in compilation techniques for functional programming languages. STELLA's extension system may serve as a basis for studying different compiler backends. In particular, while in our course we touch on STG [17], other intermediate representations such as GRIN [8, 7] and HVM [24, 20] deserve attention. However, more extensions related to operational semantics should be considered for such an exploration. We feel that this belongs to a separate course.

Another improvement to the course could be to improve the quality and automation of tests. Instead of relying merely on the exit code of the type checker, we should also take into account the type error message, which should follow a standard specified in STELLA documentation and available in the playground. A partial implementation of that idea is given by error tags, which are provided in Stella output. Additionally, in some situations, the playground is able to suggest some alternative errors, allowing for some variation in student's implementations. An approach to automation of testing similar to Lübke, Fuger, Bahnsen, Billerbeck, and Schupp [19] could also be employed in the future.

Acknowledgements. We thank Artem Murashko, Timur Iakshigulov, Alexandr Kudasov, Iskander Nafikov, and Danila Korneenko for their help with Swift, Go, Rust, and Kotlin templates. We thank Asem Abdelhady for his contribution to the test suite.

References

[1] Alexander Aiken (1996): *Cool: A Portable Project for Teaching Compiler Construction*. *SIGPLAN Not.* 31(7), p. 19–24, doi:10.1145/381841.381847.

[2] Andrew W. Appel (2004): *Modern Compiler Implementation in ML*. Cambridge University Press, USA.

[3] Andrew W. Appel & Maia Ginsburg (2004): *Modern Compiler Implementation in C*. Cambridge University Press, USA.

[4] Andrew W. Appel & Jens Palsberg (2003): *Modern Compiler Implementation in Java*, 2nd edition. Cambridge University Press, USA.

[5] Bill Appelbe (1979): *Teaching Compiler Development*. In: *Proceedings of the Tenth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '79, Association for Computing Machinery, New York, NY, USA, p. 23–27, doi:10.1145/800126.809546.

[6] Daniil Berezun & Dmitry Boulytchev (2022): *Reimplementing the Wheel: Teaching Compilers with a Small Self-Contained One*. In Peter Achten & Elena Machkasova, editors: *Proceedings Tenth and Eleventh International Workshop on Trends in Functional Programming In Education, TFPIE 2021 / 2022, Kraków, Poland (online), 16th February 2021 / 16th March 2022*, *EPTCS* 363, pp. 22–43, doi:10.4204/EPTCS.363.2.

[7] Urban Boquist (1999): *Code optimization techniques for lazy functional languages*. Ph.D. thesis, Chalmers Tekniska Högskola.

[8] Urban Boquist & Thomas Johnsson (1997): *The GRIN project: A highly optimising back end for lazy functional languages*. In: *Implementation of Functional Languages: 8th International Workshop, IFL'96 Bad Godesberg, Germany, September 16–18, 1996 Selected Papers 8*, Springer, pp. 58–84, doi:10.1007/3-540-63237-9_19.

[9] Edwin Brady (2017): *Type-driven development with Idris*. Simon and Schuster.

[10] Thierry Coquand & Gérard Huet (1988): *The calculus of constructions*. *Information and Computation* 76(2), pp. 95–120, doi:10.1016/0890-5401(88)90005-3.

[11] Luís Damas (1984): *Type assignment in programming languages*. Ph.D. thesis, University of Edinburgh, UK. Available at https://hdl.handle.net/1842/13555.

[12] Moritz Eysholdt & Heiko Behrens (2010): *Xtext: implement your language faster than the quick and dirty way*. In: *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 307–309, doi:10.1145/1869542.1869625.

[13] Markus Forsberg & Aarne Ranta (2004): *BNF Converter*. In: *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, Association for Computing Machinery, New York, NY, USA, p. 94–95, doi:10.1145/1017472.1017475.

[14] Jean-Yves Girard (1986): *The system F of variable types, fifteen years later*. *Theoretical Computer Science* 45, pp. 159–192, doi:10.1016/0304-3975(86)90044-7.

[15] R. Hindley (1969): *The Principal Type-Scheme of an Object in Combinatory Logic*. *Transactions of the American Mathematical Society* 146, pp. 29–60, doi:10.1090/S0002-9947-1969-0253905-6. Available at http://www.jstor.org/stable/1995158.

[16] Atsushi Igarashi, Benjamin C. Pierce & Philip Wadler (2001): *Featherweight Java: A Minimal Core Calculus for Java and GJ*. *ACM Trans. Program. Lang. Syst.* 23(3), p. 396–450, doi:10.1145/503502.503505.

[17] Simon L. Peyton Jones (1992): *Implementing Lazy Functional Languages on Stock Hardware: The Spineless Tagless G-Machine*. *J. Funct. Program.* 2(2), pp. 127–202, doi:10.1017/S0956796800000319.

[18] Neelakantan R. Krishnaswami (2009): *Focusing on pattern matching*. In Zhong Shao & Benjamin C. Pierce, editors: *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, ACM, pp. 366–378, doi:10.1145/1480881.1480927.

[19] Ole Lübke, Konrad Fuger, Fin Hendrik Bahnsen, Katrin Billerbeck & Sibylle Schupp (2023): *How to Derive an Electronic Functional Programming Exam from a Paper Exam with Proofs and Programming Tasks*. In: *Trends in Functional Programming in Education (TFPIE)*.

[20] Victor Maia (2023): *Higher-order Virtual Machine (HVM)*. Available at `https://github.com/HigherOrderCO/hvm`.

[21] Luc Maranget (2007): *Warnings for pattern matching*. *Journal of Functional Programming* 17(3), p. 387–421, doi:`10.1017/S0956796807006223`.

[22] Per Martin-Löf & Giovanni Sambin (1984): *Intuitionistic type theory*. 9, Bibliopolis Naples.

[23] Nicholas D. Matsakis & Felix S. Klock (2014): *The Rust Language*. In: *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, Association for Computing Machinery, New York, NY, USA, p. 103–104, doi:`10.1145/2663171.2663188`.

[24] Damiano Mazza (2007): *A denotational semantics for the symmetric interaction combinators*. *Mathematical Structures in Computer Science* 17(3), pp. 527–562, doi:`10.1017/S0960129507006135`.

[25] M. Mernik & V. Zumer (2003): *An educational tool for teaching compiler construction*. *IEEE Transactions on Education* 46(1), pp. 61–68, doi:`10.1109/TE.2002.808277`.

[26] Robin Milner (1978): *A theory of type polymorphism in programming*. *Journal of Computer and System Sciences* 17(3), pp. 348–375, doi:`10.1016/0022-0000(78)90014-4`.

[27] Steven S. Muchnick (1998): *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[28] Victor Nazarov, Hamish Mackenzie & Luite Stegeman (2015): *GHCJS Haskell to JavaScript compiler*. Available at `https://github.com/ghcjs/ghcjs`.

[29] Francisco Ortin, Daniel Zapico & Juan Manuel Cueva (2007): *Design Patterns for Teaching Type Checking in a Compiler Construction Course*. *IEEE Transactions on Education* 50(3), pp. 273–283, doi:`10.1109/TE.2007.901983`.

[30] Johan Östlund & Tobias Wrigstad (2010): *Welterweight Java*. In: *Proceedings of the 48th International Conference on Objects, Models, Components, Patterns*, TOOLS'10, Springer-Verlag, Berlin, Heidelberg, p. 97–116, doi:`10.1007/978-3-642-13953-6_6`.

[31] Benjamin C. Pierce (2002): *Types and programming languages*. MIT Press.

[32] Gordon D. Plotkin (1977): *LCF Considered as a Programming Language*. *Theor. Comput. Sci.* 5, pp. 223–255, doi:`10.1016/0304-3975(77)90044-5`. Available at `https://api.semanticscholar.org/CorpusID:53785015`.

[33] John C Reynolds (1974): *Towards a theory of type structure*. In: *Programming Symposium: Proceedings, Colloque sur la Programmation Paris, April 9–11, 1974*, Springer, pp. 408–425, doi:`10.1007/3-540-06859-7_148`.

[34] Peter Sestoft (1996): *ML Pattern Match Compilation and Partial Evaluation*. In Olivier Danvy, Robert Glück & Peter Thiemann, editors: *Partial Evaluation, International Seminar, Dagstuhl Castle, Germany, February 12-16, 1996, Selected Papers*, *Lecture Notes in Computer Science* 1110, Springer, pp. 446–464, doi:`10.1007/3-540-61580-6_22`.

[35] M. Wand (1989): *Type inference for record concatenation and multiple inheritance*. In: *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pp. 92–97, doi:`10.1109/LICS.1989.39162`.

[36] Richard Wei, Dan Zheng, Marc Rasi & Bart Chrzaszcz (2023): *Differentiable Programming Manifesto*. Available at `https://github.com/apple/swift/blob/main/docs/DifferentiableProgramming.md`.

[37] Niklaus Wirth (1996): *Compiler Construction*. Addison Wesley Longman Publishing Co., Inc., USA.

[38] Brent A. Yorgey (2023): *Disco: A Functional Programming Language for Discrete Mathematics*. *Electronic Proceedings in Theoretical Computer Science* 382, p. 64–81, doi:`10.4204/eptcs.382.4`.

Functional Programming in Learning Electromagnetic Theory

Scott N. Walck
Department of Physics
Lebanon Valley College
Annville, Pennsylvania, USA
walck@lvc.edu

Electromagnetic theory is central to physics. An undergraduate major in physics typically takes a semester or a year of electromagnetic theory as a junior or senior, and a graduate student in physics typically takes an additional semester or year at a more advanced level. In fall 2023, the author taught his undergraduate electricity and magnetism class using numerical methods in Haskell in parallel with traditional analytical methods. This article describes what functional programming has to offer to physics in general, and electromagnetic theory in particular. We give examples from vector calculus, the mathematical language in which electromagnetic theory is expressed, and electromagnetic theory itself.

1 Introduction

James Clerk Maxwell put the finishing touches on modern electromagnetic theory in 1865, publishing the famous equations that still bear his name. It is the oldest piece of theoretical physics that is part of humanity's current best understanding of the physical world.

Electromagnetic (EM) theory describes one of the four fundamental forces of nature. Today's physicists believe that all known interactions can be classified as one of four forces. These are the strong force, the electromagnetic force, the weak force, and gravity. Electricity holds atoms together, which seems enough to give it a claim to importance.

Electromagnetic theory serves as the model for modern field theories of elementary particles. EM theory is the prototypical example of a *field theory*, that is, a theory whose important quantities depend on space or spacetime. Modern theories of elementary particle physics are based on quantum field theories, the quantum versions of theories like electromagnetic theory.

Surprisingly, Maxwell's 1865 electromagnetic theory did not need to be modified to incorporate Einstein's 1905 relativity theory or the quantum theory of 1925. In the case of relativity theory, electromagnetic waves began to be interpreted differently, as physicists discarded the notion of an ether in which EM waves traveled. In the case of quantum theory, mathematical objects like electric field and magnetic field were interpreted differently, as operators rather than numbers or vectors, but the Maxwell equations remained intact. It seems that Faraday and Maxwell were onto something fundamental about the universe.

Electromagnetic theory is the earliest theory that is still part of our current best understanding of the universe. Newtonian mechanics is incredibly useful, and beautiful, but it cannot be said to express our current best ideas about the universe. The 20th-century ideas of quantum mechanics and relativity have each led to newer theories that are slightly different from Newtonian mechanics, although they give essentially the same answers for slow massive things. Electromagnetic theory, on the other hand, passed unchanged through the 20th-century quantum and relativity revolutions.

Stephen Chang (Ed.): Trends in Functional
Programming in Education (TFPIE)
EPTCS 405, 2024, pp. 20–35, doi:10.4204/EPTCS.405.2

© Scott N. Walck
This work is licensed under the
Creative Commons Attribution License.

Electromagnetic theory unites electricity, magnetism, and light into a single theory. Light is an electromagnetic wave, a wave of electric and magnetic fields. Electromagnetic theory is three theories in one, describing electricity, magnetism, and optics.

We take the attitude of Papert[3] and others[4, 5, 6, 7] that students are aided in their learning by having building blocks with which to create interesting structures, that such creative activity is a motivating and effective way to learn, and that the feedback provided by computer-language-based building blocks can expose our confusions and produce delight in our achievements.

The author has written previously about the benefits of functional programming to physics. In [6], we showed how Newtonian mechanics benefits from expression in a functional programming language. In [7], we showed how a course in quantum mechanics can benefit from functional programming. The textbook [8] treats both undergraduate Newtonian mechanics and electromagnetic theory. It can be used as a textbook for a course in computational physics or as a companion text for courses in classical mechanics or electromagnetism. The author of this paper uses [8] in both ways. The present paper focuses on functional programming in electromagnetic theory, and in particular on the author's experience incorporating it into a traditional undergraduate electromagnetic theory course.

The plan for the paper is as follows. In Section 2, we describe what functional programming has to offer to physics in general and to electromagnetic theory in particular. Section 3 describes field theory and how it can be encoded in Haskell. In Section 4, we give examples of what vector calculus can look like in a functional programming language. Section 5 describes an example of electromagnetic theory, namely how magnetic field is produced by electric current.

2 What Does Functional Programming Offer to Physics Pedagogy?

In this section, we describe how programming in general, and functional programming in particular, can help people learn physics, especially electromagnetic theory. The first three items deal with programming in general, while the last five focus on functional programming.

2.1 With a little programming, we can study physical situations that are not exactly solvable.

Most physical situations are not exactly solvable. The traditional tools of algebra, calculus, and differential equations only get us so far. They can't solve most problems. The theory tells you how to make (partial) differential equations, but you can't solve them exactly. We don't know how to write down functions that exactly satisfy the differential equations. Traditionally, you spend a lot of time studying situations that *can* be solved exactly, and then come up with tricks to approximate the other situations that can't be solved exactly.

In EM theory, the electric field inside a capacitor with infinite plates is exactly solvable, but the electric field inside a capacitor with finite plates is not. The magnetic field produced by a circular loop carrying current is exactly solvable along the axis that runs through the center of the circle (perpendicular to the plane of the circle), but not exactly solvable anywhere else.

There are some simple, standard approximations (numerical methods), that are known to be reasonably good, and don't involve tricks. The problem is that they require an enormous number of exceedingly boring computations. And here is where the computer comes to the rescue. Just by being able and willing to do an enormous number of simple, boring computations, the computer can give us good, approximate solutions to many problems.

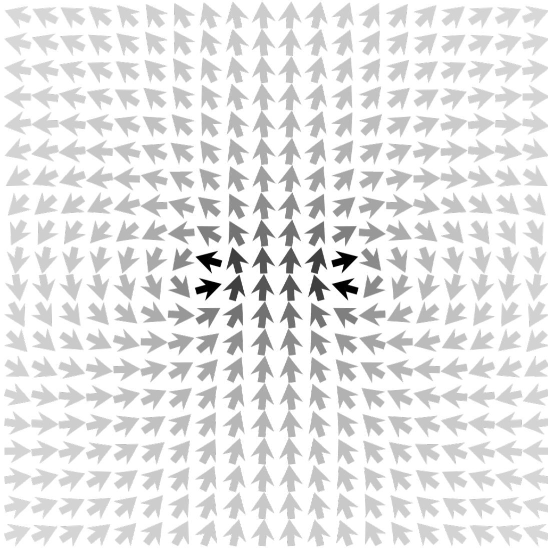


Figure 1: Magnetic field produced by a current loop. The current loop lies in the *xy* plane. The figure shows the *yz* plane. A darker arrow indicates a stronger magnetic field. The current loop (not shown) pierces the *yz* plane near the darkest arrows. This situation, simple as it seems, is exactly solvable only on the *z* axis, running vertically through the center of the figure.

Figure 1 shows the numerically calculated magnetic field of a circular current loop, a problem which, surprisingly, is not analytically solvable.

2.2 Computer programming is a modern tool that can help solve problems.

Why not use all the tools at our disposal, and in particular the computer, to help us solve our problems? Computers can do things that calculators can't. I don't predict the death of the pocket calculator, but who knows? The slide rule is, if not dead, an eclectic tool used by very few people.

Physics societies like the American Association of Physics Teachers (AAPT) have been encouraging physics teachers to include more computer techniques in their courses.

2.3 Programming is a valuable skill in itself.

Programming, like mathematics and like writing, is a valuable skill in itself. Knowing something about programming is useful for getting a job, but it's also useful for organizing your thinking. (Mathematics and writing also help us organize our thinking.) Writing code is not just about getting the computer to do something. It's also about expressing ideas using formal language in a way that makes sense to people. And so, for the same reason that essay writing can help you clarify your ideas about art or politics, code writing can help you clarify your ideas about physics.

2.4 Writing in a language that will be read by a computer forces a precision that human language and mathematical notation sometimes lack.

In the preface to their wonderful book *Structure and Interpretation of Classical Mechanics* (SICM)[4], Sussman and Wisdom note

> The traditional use of ambiguous notation is convenient in simple situations, but in more complicated situations it can be a serious handicap to clear reasoning. In order that the reasoning be clear and unambiguous, we have adopted a more precise mathematical notation. Our notation is functional and follows that of modern mathematical presentations.

and then

> We require that our mathematical notations be explicit and precise enough that they can be interpreted automatically, as by a computer.

Sussman was the first pioneer in the application of functional programming to physics education. He showed the power of higher-order functions in physics using *Scheme*, a dialect of Lisp he co-invented with Steele, and used with Abelson in their classic computer science text *Structure and Interpretation of Computer Programs*[1].

The approach described in this paper is similar to SICM in that both are interested in using functional programming as a language for physics. Our approach differs from SICM in several ways. First, we use Haskell instead of Scheme, a difference which is noticeable primarily in our reliance on types and Haskell's type system. Second, SICM is a graduate-level textbook while we target undergraduates. Third, SICM restricts its attention to classical mechanics, the theory of motion created by Newton and developed by Lagrange and Hamilton, while our concern in this paper is with a different theory of physics, namely electromagnetic theory.

2.5 Avoiding mutation makes a programming language closer to mathematics.

The core of a pure functional programming language, while providing many interesting challenges for the compiler designer, presents a very simple model of computation to the user of the language. Names stand for values, not memory locations. Names refer to a single thing, and don't change over time.

These are the principles upon which mathematical notation is based. Physics already uses mathematical language, so it is a smaller step for the physics student to learn the core of a pure functional programming language than it is to learn an imperative language. Obviously, there are trade-offs; functional programming's affection for recursion can make teaching iteration more difficult.

In my courses, I teach students how to use the Haskell functions `map`, `iterate`, `filter`, `take`, and `takeWhile` as well as list comprehensions, but I do not teach them how to write explicitly recursive functions. Teaching recursion is a substantial endeavor, and I need to use the time to teach ideas of physics.

2.6 Types in a language help the reader understand and the writer clarify and organize.

I have come to believe that a strong system of types is exceedingly helpful in a language for physics. Algebraic data types are useful in a language for physics for the same reason they are helpful in any domain-specific language, namely that the key ideas of the domain can be encoded as types.

The code writer can experience the benefits of clarity and organization of types even if she is not the person writing the data type definition.

An example of an algebraic data type in physics is *charge distribution*. Physicists like to talk about point charges, but they also like to talk about electric charge that is distributed along a curve, across a surface, or throughout a volume. A *charge distribution* is a specification of electric charge that could be at a point, along a curve, across a surface, throughout a volume, or some combination of these.

We use an algebraic data type to make charge distribution into a type.

```
data ChargeDistribution
    = PointCharge   Charge      Position
    | LineCharge    ScalarField Curve
    | SurfaceCharge ScalarField Surface
    | VolumeCharge  ScalarField Volume
    | MultipleCharges [ChargeDistribution]
```

The definition tells us that a charge distribution could be a point charge, a line charge (physicists use the term *line charge* for any one-dimensional distribution of charge; it need not occur along a straight line), a surface charge, a volume charge, or a combination of these. The `ChargeDistribution` data type is algebraic in having multiple constructors, and recursive in the last constructor.

We can see from its definition that the `ChargeDistribution` data type is based on other data types for physics like `Charge`, `Position`, `ScalarField`, `Curve`, `Surface`, and `Volume`. `Charge` is just a type synonym for double-precision floating point number. `Position` is a point in three-dimensional space. We'll discuss the `Position` data type in the next section. `Curve`, `Surface`, and `Volume` are geometric objects that describe finite curves, surfaces, and volumes. These are helpful because EM theory has a substantial geometric content.

I do not demand that students be able to write algebraic data type definitions like this. Some students are interested in how this works, and I'm always happy to share what I know, but my main interest is in having students use this data type to describe a charge distribution. Using this data type brings one face to face with precisely what a charge distribution is, how we talk about it, and what's it's good for.

In EM theory, scalar and vector fields play a prominent role. A *field* in physics is a function of space or spacetime. In an undergraduate setting, it is more common to treat fields as functions of space, and I follow that custom with the following type synonyms for scalar field and vector field.

```
type ScalarField = Position -> R
type VectorField = Position -> Vec
```

In the first definition, `R` is a type synonym for `Double`, a double-precision floating point number. I like to think of these numbers as approximations to real numbers, and they occur so often in physics that it is convenient to give them a shorter name that comes closer to how we think about them. In the second definition, `Vec` is a data type for three-dimensional vectors. Such vectors are ubiquitous in physics, especially at the undergraduate pre-relativity level.

The central idea of electrostatics, the portion of EM theory focusing on electrical phenomena when nothing is changing in time, is that a charge distribution produces an electric field. We encode this central idea as a function.

```
eField :: ChargeDistribution -> VectorField
```

The type signature of `eField` sings the central idea of electrostatics. The definition of `eField` takes some effort to develop, as we need various kinds of integrals in order to deal with the continuous charge distributions. The type signature, however, serves as a wonderful summary and reminder of the big picture, and hopefully serves as a counterbalance to the possibility that people might get lost in the details of the function definition.

In fact, students routinely get lost in the traditional analytical presentation of electric field produced by a line charge, electric field produced by a surface charge, and electric field produced by a volume charge. There are so many details to worry about, and nothing in the traditional mathematical notation is reminding us of the central idea. A language with a strong system of types can help us keep the important ideas in mind.

2.7 Higher-order functions are a natural way to express the higher-order ideas that physics trades in.

Take the derivative from calculus as a higher-order idea that physics uses. The derivative of a function at a point expresses the rate at which the function changes at that point. We can write a numerical derivative in Haskell.

```
type R = Double

derivative :: R -> (R -> R) -> R -> R
derivative dt f t = (f (t + dt/2) - f (t - dt/2)) / dt
```

In this definition, `dt` is a numerical step size for the numerical derivative we are calculating. The second argument is the function `f :: R -> R` we want to differentiate. The third argument `t :: R` is the value of the independent variable where we want to estimate the slope.

Physicists like to think of the derivative as an "operator" that takes a function as input and gives another function as output. Haskell's currying allows us to think just like that. Instead of thinking of the `derivative` as a function that takes three inputs (step size, function, and independent variable), we can think of `derivative` as a function that takes two inputs (step size and function) and returns a function. Even better, we can come to see that these two ways of thinking are the same for the computer because there is a sense in which there is no difference. The two ways of thinking are equivalent. Better yet, the function `derivative 1e-6` really is the "operator" that takes a function as input and gives a function as output.

Numeric integration is another example of a higher-order function that encodes a basic idea of physics. And since calculus, vector calculus, and differential equations are so ubiquitous in physics, there are a plethora of higher-order functions serving calculus-related roles, some of which we will see later in this article.

Electromagnetic theory is primarily about vector fields, namely the electric and magnetic fields. We saw above that a vector field is a function, so any function that deals with a vector field is a higher-order function.

In physics, ideas we regard as basic, like the derivative, the integral, and the vector field, naturally invite us to use higher-order functions. Because higher-order functions in Haskell and other functional programming languages are easy to write and use, the language of the code can help you understand the theory. When a language allows a succinct expression of an idea that a domain regards as basic, using that language allows one to experiment and play with the idea, which is a great way to learn. Reading and writing in a language that makes it easy (or at least easier) to express the ideas in a theory promotes understanding of the theory. I claim that typed functional languages are especially suitable for physics, and I show examples below as evidence of this claim. I have anecdotal evidence that students understand analytical methods in electromagnetic theory better for having studied numerical methods in a functional programming language.

2.8 Functional programming offers a vision of a language for both calculating and proving, two activities that theoretical physics is full of.

Physics has some results that can be obtained from deductive reasoning. The Maxwell equations imply conservation of electric charge, for example. In a course in EM theory, we show how to derive the continuity equation, an expression of local charge conservation, from the Maxwell equations.

This derivation takes a handful of steps, and although it doesn't rise to the stature of a theorem as appreciated by mathematicians, nevertheless the language and pattern of a theorem are present. If the Maxwell equations are true, then the continuity equation is true. Such relationships are helpful to recognize, because a deep understanding of any theory requires going beyond "what is true" into knowing when ideas are independent, and knowing when one idea is a logical consequence of another.

Haskell, the language that I use in my EM theory class, is great for calculation, but not really intended for theorem proving. However, a handful of functional languages based on dependent types, such as Coq, Agda, Idris, and Lean, are capable of theorem proving (meaning proof checking with some automation) in addition to calculation.

It seems that a language that could calculate and prove would be a wonderful language in which to express the principles of physics. While I have great interest in this possibility, it is in no sense a reality in my teaching.

It seems to me that the possibility of using a dependently-typed functional language for physics in the future is a reason to learn a functional language like Haskell for physics now.

3 Field Theory in Haskell

Electromagnetic theory is a *field theory*, where a *field* in physics is a quantity that depends on location in space or spacetime. (The definition of *field* in mathematics is different, and unrelated.) The most common quantities that depend on space or spacetime are scalars (basically numbers) and vectors. In undergraduate electromagnetic theory, it is common to view fields as functions of space, and we will follow this practice. A scalar quantity that depends on space is called a *scalar field*, and a vector quantity that depends on space is called a *vector field*. We gave type definitions for `ScalarField` and `VectorField` in section 2.6.

EM theory can be expressed (quite beautifully, in fact) in relativistic notation where the vectors are 4-vectors (elements of a 4-dimensional vector space) and the fields are functions of spacetime, but in an undergraduate setting, it is much more common to use a notation in which vectors are 3-vectors (members of a 3-dimensional vector space), and the fields are functions of space. This does not preclude the possibility that fields can change in time; it merely places space and time on an apparently different footing.

Having decided to pursue an undergraduate notation in which 3-dimensional space is central, we turn to the question of how to represent points in 3-dimensional space. Our first choice is Cartesian coordinates x, y, z, in which the three coordinate axes are mutually perpendicular. However, several important situations in EM theory have spherical or cylindrical symmetry, which motivates the use of spherical and cylindrical coordinates. Even situations which do not have full cylindrical or spherical symmetry can benefit from these coordinates. In fact, there are many possible coordinate systems for 3-dimensional space, but we will focus on the 3 most commonly used, namely Cartesian, cylindrical, and spherical. Figure 2 shows how cylindrical and spherical coordinates are defined.

In cylindrical coordinates, we use s to represent the distance from the z axis, ϕ to represent the angle shown in Figure 2, and z to mean the same thing it means in Cartesian coordinates. The cylindrical coordinates of a point in space are then given by the triple (s, ϕ, z). In spherical coordinates, we use r to represent the distance from the origin, θ to represent the angle shown in Figure 2, and ϕ to mean the same thing it means in cylindrical coordinates. The spherical coordinates of a point in space are then given by the triple (r, θ, ϕ). This notation for cylindrical and spherical coordinates comes from Griffiths' popular textbook[2].

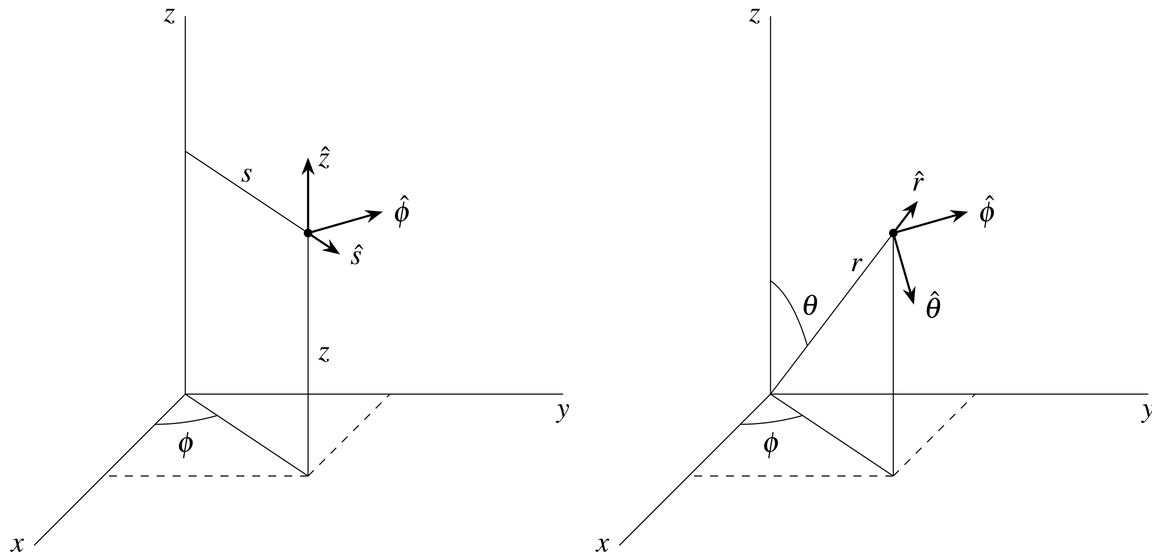


Figure 2: Cylindrical coordinates (left) and spherical coordinates (right)

The purpose of the `Position` data type is to free our minds from the detail of establishing a single, canonical way of referring to a point in space. We wish to freely use any of the three coordinate systems in constructing a `Position`, and any of the three in extracting coordinates. Here is the definition of the `Position` data type.

```
data Position = Cart R R R
                deriving (Show)
```

We store coordinates in Cartesian form, and name the constructor `Cart` to remind us of this fact, but users of `Position` need not be aware of the specifics of how the coordinates are stored.

The following three functions construct a `Position` from coordinates.

```
cartesian   :: (R,R,R) -> Position   -- (x,y,z)
cylindrical :: (R,R,R) -> Position   -- (s,phi,z)
spherical   :: (R,R,R) -> Position   -- (r,theta,phi)
```

For convenience, we define curried forms `cart`, `cyl`, and `sph`.

Finally, we have three functions to extract coordinates from a `Position` in any of the three systems.

```
cartesianCoordinates   :: Position -> (R,R,R)   -- (x,y,z)
cylindricalCoordinates :: Position -> (R,R,R)   -- (s,phi,z)
sphericalCoordinates   :: Position -> (R,R,R)   -- (r,theta,phi)
```

Since the `VectorField` is the type of the electric field and the magnetic field, and since we can develop some geometric intuition by making pictures of vectors fields, we spend some time doing this in my class. The function `vfGrad` uses a technique I call *gradient visualization*, in which the strength of the field is denoted by the darkness of arrows. This is the kind of visualization used in Figure 1 for the magnetic field produced by a circular current loop.

```
vfGrad :: (R -> R)
       -> ((R,R) -> Position)
```

```
-> (Vec -> (R,R))
-> FilePath
-> Int    -- n for n x n
-> VectorField
-> IO ()
```

There are several parameters that the function `vfGrad` needs in order to produce a picture like Figure 1. The first argument to `vfGrad` is a monotonic function that allows the transition from light arrows to dark arrows to occur in a nonlinear way; using this we can avoid pictures with a small number of black arrows on an otherwise white background that would occur with a linear scaling because the field is very strong at one location. The next two arguments to `vfGrad` control the relationship between the 3-dimensional field and the 2-dimensional picture; they specify which plane we are looking at and how to convert 3-dimensional vectors into 2-dimensional arrows. The last three arguments to `vfGrad` are a file name for the graphics file, an integer describing how many arrows we want in each direction, and finally the vector field itself. The definition of this function uses Brent Yorgey's *diagrams* package[9].

With types for scalar fields and vector fields, we are ready to explore the mathematics used in electromagnetic theory, vector calculus.

4 Vector Calculus in Haskell

Vector calculus is a large subject. Undergraduate physics majors at Lebanon Valley College take one semester of vector calculus, but that is not quite enough for them to get to results like Stokes' theorem, which we need in electromagnetic theory. So, part of our electricity and magnetism course is a review and deeper dive into vector calculus.

I have always included some vector calculus in my electricity and magnetism course, but in fall 2023, I taught analytical methods for vector calculus in parallel with numerical methods using Haskell. This section gives a sampling of what vector calculus can look like in Haskell.

In order to do vector calculus, we are first going to need a data type for vectors. The `Vec` type was seen in passing in Section 2.6, and is for 3-dimensional vectors. Details about the `Vec` type can be found in [6] and [8].

Electromagnetic theory is a geometric subject, and the vector calculus it uses has a geometric flavor. To see why, consider the following form of the Maxwell equations.

4.1 Maxwell Equations, Integral Form

There are several ways of writing the Maxwell equations. Below we give a version known as the "integral form". (There is also a "differential form" that is probably seen more often, but hides the geometric character of the theory.) For each of the four equations, I give a statement in words, followed by an equation in mathematical notation that expresses the same thing.

1. **Ampere-Maxwell Law:** The rate of change of electric flux through a surface is the magnetic circulation around its boundary minus the current flowing through the surface.

$$\frac{d}{dt}\int_S \vec{E}\cdot d\vec{a} = \frac{1}{\varepsilon_0\mu_0}\int_{\partial S}\vec{B}\cdot d\vec{\ell} - \frac{1}{\varepsilon_0}I$$

2. **Faraday's Law:** The rate of change of magnetic flux through a surface is the opposite of the electric circulation around its boundary.

$$\frac{d}{dt}\int_S \vec{B}\cdot d\vec{a} = -\int_{\partial S} \vec{E}\cdot d\vec{\ell}$$

3. **Gauss's Law:** The electric flux through a closed surface is the charge contained inside.

$$\int_S \vec{E}\cdot d\vec{a} = \frac{1}{\varepsilon_0}Q$$

4. **No Magnetic Monopoles Law:** The magnetic flux through a closed surface is zero.

$$\int_S \vec{B}\cdot d\vec{a} = 0$$

In these equations, $\vec{E}$ is the electric field and $\vec{B}$ is the magnetic field. Electric and magnetic fields are the fundamental physical quantities that electromagnetic theory deals with. They are created by charged particles, and they exert forces on charged particles. Most (perhaps all, I have no counterexample) physicists regard electric and magnetic fields to be just as real as the particles they interact with. Particles and fields, then, have a symbiotic relationship. Conceptually, they are equal partners in the explanation of electromagnetic phenomena, both dynamic actors on the electromagnetic stage. Most people, students included, have a much easier time imagining a particle than imagining a field. Electric and magnetic fields have type `VectorField`, which is a type synonym for `Position -> Vec`.

In the first two equations, S is a surface and ∂S is its boundary curve. An electric current I flows through surface S. In the third and fourth equations, S is a closed surface, containing some volume, and Q is the total electric charge in that volume. The constants ε_0 and μ_0 appear in the equations because they use the International System (SI) of units. These constants are elided from the English-language descriptions.

Notice that surfaces and their boundary curves make an explicit appearance in this form of the Maxwell equations. This is evidence of the geometrical character of electromagnetic theory.

Beyond their role in the statement of the Maxwell equations, curves and surfaces are used to describe continuous charge and current distributions. We saw an example of this with the definition of the `ChargeDistribution` type in Section 2.6. The `ChargeDistribution` data type makes reference to the types `Curve`, `Surface`, and `Volume`. The simplest of these is the `Curve` data type, defined as follows.

```
data Curve = Curve { curveFunc          :: R -> Position
                   , startingCurveParam :: R  -- t_a
                   , endingCurveParam   :: R  -- t_b
                   }
```

A `Curve` is a map of one parameter into space, along with initial and final values of that parameter. A `Surface` is a map of two parameters into space, along with boundary data.

```
data Surface = Surface ((R, R) -> Position) R R (R -> R) (R -> R)
```

The first argument to `Surface` is the parameterizing function that maps (s,t) into a `Position`. The second and third arguments are lower and upper limits of the parameter s. The fourth and fifth arguments are functions of s that give the lower and upper limits of t. Allowing the limits of the second parameter to depend on the first parameter makes it easier to describe a surface like a triangle. The boundary of a `Surface` is a `Curve`.

A `Volume` is a map of three parameters into space, along with boundary information. The boundary of a `Volume` is a `Surface`.

Once we have data types for curves, surfaces, and volumes, vector calculus wants us to perform integrals over them.

4.2 Flux and Circulation Integrals

For any vector field $\vec{F}$ and any surface S, the *flux* of the field through the surface is the integral

$$\int_S \vec{F} \cdot d\vec{a}.$$

The integral is calculated by approximating the surface S by many small triangles. Each triangle has a vector area whose magnitude is the area of the triangle and whose direction is perpendicular to the triangle. At each triangle, we form the inner (dot) product of the vector field $\vec{F}$ at the location of the triangle and the vector area of the triangle; this inner product is a number. We add up these numbers for all of the triangles and call the result the flux of $\vec{F}$ through S. If the vector field is the electric field, we call the flux electric flux. If the vector field is the magnetic field, we call the flux magnetic flux. If $\vec{F}$ represented the flow of stuff, then the flux of $\vec{F}$ through S is how much stuff flows through S.

Here is the Haskell code to approximate a flux integral.

```
dottedSurfaceIntegral :: SurfaceApprox -> VectorField -> Surface -> R
dottedSurfaceIntegral approx vF s
    = sum [vF r' <.> da' | (r',da') <- approx s]
```

The function `dottedSurfaceIntegral` needs an approximation method (type `SurfaceApprox`) to divide the surface into triangles.

```
type SurfaceApprox = Surface -> [(Position, Vec)]
```

This approximation method, called `approx` in the code, produces a list of pairs of positions and vector areas. The left side of the list comprehension shows that we take the dot product (using the operator `<.>`) of the vector field `vF` evaluated at position `r'` with the vector area `da'`. The numerical results of the inner products are then added up with `sum`.

For any vector field $\vec{F}$ and any curve C, the *circulation* of the field along the curve is the integral

$$\int_C \vec{F} \cdot d\vec{\ell}.$$

The integral is calculated by approximating the surface C by many small segments. Each segment has a vector length whose magnitude is the length of the segment and whose direction is along the segment. At each segment, we form the inner (dot) product of the vector field $\vec{F}$ at the location of the segment and the vector length of the segment; this inner product is a number. We add up these numbers for all of the segments and call the result the circulation of $\vec{F}$ along C. If the vector field is the electric field, we call the circulation electric circulation. If the vector field is the magnetic field, we call the circulation magnetic circulation.

Here is the Haskell code to approximate a circulation integral.

```
dottedLineIntegral :: CurveApprox -> VectorField -> Curve -> R
dottedLineIntegral approx f c = sum [f r' <.> dl' | (r',dl') <- approx c]
```

The function `dottedLineIntegral` needs an approximation method to divide the curve into segments. This approximation method, called `approx` in the code, produces a list of pairs of positions and vector lengths. The left side of the list comprehension shows that we take the dot product (using the operator `<.>`) of the vector field `f` evaluated at position `r'` with the vector length `dl'`. The numerical results of the inner products are then added up with `sum`.

There are quite a number of these line, surface, and volume integrals that are used in electromagnetic theory. The table below shows the nine integrals that EM theory needs from vector calculus.

Haskell function	mathematical notation	requires	
`scalarLineIntegral`	$\int_C f\, d\ell$	scalar field	curve
`vectorLineIntegral`	$\int_C \vec{F}\, d\ell$	vector field	curve
`dottedLineIntegral`	$\int_C \vec{F} \cdot d\vec{\ell}$	vector field	curve
`crossedLineIntegral`	$\int_C \vec{F} \times d\vec{\ell}$	vector field	curve
`scalarSurfaceIntegral`	$\int_S f\, da$	scalar field	surface
`vectorSurfaceIntegral`	$\int_S \vec{F}\, da$	vector field	surface
`dottedSurfaceIntegral`	$\int_S \vec{F} \cdot d\vec{a}$	vector field	surface
`scalarVolumeIntegral`	$\int_V f\, dv$	scalar field	volume
`vectorVolumeIntegral`	$\int_V \vec{F}\, dv$	vector field	volume

Vector calculus has its own versions of the fundamental theorem of calculus, the theorem that makes precise the idea that derivatives and integrals are inverse operations of each other. The fundamental theorems of vector calculus are known as the gradient theorem, Stokes' theorem, and the divergence theorem. They can be expressed using the integrals in the table above.

4.3 A Homework Problem

After students understand the meaning of the integrals in the table above, they are in a position to study the three fundamental theorems of vector calculus. Since each theorem claims that an integral of a derivative over some geometric object is equal to a different integral over the boundary of that object, checking these fundamental theorems provides opportunities to practice evaluating the integrals. Even better, the two sides, which are each a bit of a challenge to set up, must give the same number in the end, which serves as satisfying evidence that the integrals were done correctly.

Stokes' theorem claims that for any vector field $\vec{F}$ and any surface S, the following equality holds.

$$\int_S (\vec{\nabla} \times \vec{F}) \cdot d\vec{a} = \int_{\partial S} \vec{F} \cdot d\vec{\ell}$$

The expression $\vec{\nabla} \times \vec{F}$ is called the curl of $\vec{F}$. The curl is a vector derivative that takes a `VectorField` as input and produces a `VectorField` as output. The physical meaning of the curl is the extent to which a vector field, if it were a velocity field of water, encourages rotation of a small object placed in the water. A vector field with zero curl is called conservative, meaning that a dotted line integral of the field around any closed curve is zero.

The following homework problem, which was assigned in fall 2023, asks students to check Stokes' theorem for a particular vector field and a particular surface.

Homework Problem: Check Stokes' theorem using the vector field

$$\vec{F}(x,y,z) = -z\hat{y} + y\hat{z}$$

and the rectangular region with corners $(x,y,z) = (0,0,-4)$, $(0,2,-4)$, $(0,2,4)$, $(0,0,4)$, in which the orientation is in the positive x direction.

Checking the fundamental theorem for curls means (a) finding the curl of your vector field, (b) finding the flux integral of the curl over the surface, (c) finding the line integral (the circulation) of your vector field over the boundary of the surface (which has 4 parts), and (d) confirming that the results from (b) and (c) are the same. You may do this problem analytically, numerically, or a combination of both.

To solve this homework problem numerically with Haskell, we first encode the vector field $\vec{F}$ as `vF` and the rectangle as `rect`.

```
vF :: VectorField
vF r = let (x,y,z) = cartesianCoordinates r
       in (-z) *^ yHat r ^+^ y *^ zHat r

rect :: Surface
rect = Surface (\(y,z) -> cart 0 y z) 0 2 (const (-4)) (const 4)
```

To calculate the left-hand side of Stokes' theorem, we need to take the curl of the vector field. As with a numerical derivative, we need to give a step size over which to evaluate the curl; we choose 10^{-6}.

```
curlF :: VectorField
curlF = curl 1e-6 vF
```

Now we can calculate the left side of Stokes's theorem.

```
leftSide :: R
leftSide = dottedSurfaceIntegral (surfaceSample 200) curlF rect
```

We get the result `32.00000000236293`. For the right side, we need the boundary curve of the reactangle.

```
boundaryParam :: R -> Position
boundaryParam t
    | t < 1      = cart 0 (2*t) (-4)
    | t < 2      = cart 0 2 (8*(t-1) - 4)
    | t < 3      = cart 0 (2 - 2 * (t-2)) 4
    | otherwise  = cart 0 0 (4 - 8 * (t-3))

boundaryOfRect :: Curve
boundaryOfRect = Curve boundaryParam 0 4

rightSide :: R
rightSide = dottedLineIntegral (curveSample 1000) vF boundaryOfRect
```

The right side gives `32.0`, which is reasonably close to the left side, and can be explained by numerical error.

In this homework problem, I give students the choice about whether to use analytic (pencil and paper) methods, numerical methods, or a combination of both. I do not always give students this choice. Sometimes I demand they use analytic methods; sometimes I demand they use numerical methods. At least one third of the time, however, I give students the choice, which they seem to appreciate.

For this homework problem in fall 2023, 5 of the 10 students who submitted it chose to do it analytically, and 5 chose to do it numerically using Haskell.

As a last example of the power of types and higher order functions in EM theory, we turn to calculation of the magnetic field by the Biot-Savart law.

5 Magnetic Field Produced by a Current Distribution

Earlier, we saw that the central idea of electrostatics is that a charge distribution produces an electric field. Analogously, the central idea of magnetostatics is that a current distribution produces a magnetic field. One can define a type `CurrentDistribution`, and a function

```
bField :: CurrentDistribution -> VectorField
```

that calculates the magnetic field produced by any current distribution.[8]

In this section, we focus on a particular type of current distribution, the one we probably have in mind when thinking of electric current, namely the current flowing through a wire. The equation for calculating the magnetic field produced by current flowing through a wire is called the Biot-Savart law, and is shown below.

$$\vec{B}(\vec{r}) = -\frac{\mu_0 I}{4\pi} \int_C \frac{\vec{r}-\vec{r}'}{|\vec{r}-\vec{r}'|^3} \times d\vec{\ell}' \tag{1}$$

In this equation, C is a curve of arbitrary shape that describes the configuration of the wire. The wire could run along a straight line, around in a circle, in a helix, or any other configuration. The integral asks the *source point* $\vec{r}'$ to take all possible values along the wire, the current at each position along the wire being a source of magnetic field. The *field point* $\vec{r}$ is the place where we want to find the magnetic field; it is a constant from the perspective of the integral. The current I is just a number, usually measured in Amperes. The symbol $d\vec{\ell}'$ represents a small section of the wire. The multiplication $\times$ is the vector cross product.

Here is the Biot-Savart law, translated into Haskell.

```
bFieldFromLineCurrent
    :: Current      -- current (in Amps)
    -> Curve
    -> VectorField  -- magnetic field (in Tesla)
bFieldFromLineCurrent i c r
    = let coeff = -mu0 * i / (4 * pi)  -- SI units
          integrand r' = d ^/ magnitude d ** 3
              where d = displacement r' r
      in coeff *^ crossedLineIntegral (curveSample 1000) integrand c
```

The type `Current` is a synonym for `R` or `Double`; it is the number of Amperes of current flowing through the wire. We define a local constant `coeff` to hold the numerical value of $-\mu_0 I/4\pi$ in SI units, and we define a local function `integrand` to hold the integrand. We want to define a local variable `d` for the displacement from `r'` to `r`, but because `r'` exists locally to the function `integrand`, the definition for `d` must occur within the definition for `integrand` and cannot be placed parallel to the definitions of `coeff` and `integrand`. We use the `crossedLineIntegral` to do the integration; this function appears in the table of vector calculus integrals in Section 4.2.

The type signature of `bFieldFromLineCurrent` makes clear, in a computer-checked way, the two inputs required to find the magnetic field: the `Current` and the `Curve` along which the current flows. For a reader of Haskell, the function `bFieldFromLineCurrent` is a clearer description of what is going on than Equation 1 since the latter does not make it terribly clear that the magnetic field depends only on the curve and the current.

The function `bFieldFromLineCurrent` was used to produce Figure 1 in Section 2.1.

Next we describe a homework problem which requires the Biot-Savart law. In Section 2.1, we mentioned that the magnetic field produced by a circular loop carrying current is exactly solvable along the

axis that runs through the center of the circle (perpendicular to the plane of the circle), but not exactly solvable anywhere else. This homework problem asks students to use analytical methods to find the magnetic field on the axis, where a closed-form algebraic result can be obtained, and then to use numerical methods to extend their solution to any other point in space.

> **Homework Problem:** Find the magnetic field on the z-axis produced by a circular current loop with radius R lying in the xy-plane, centered at the origin, carrying current I. Do this analytically, then show how to use Haskell to find the magnetic field anywhere.

6 Conclusion

We've shown a number of features that functional programming languages have, especially types, higher-order functions, and referential transparency, that make them particularly appropriate for expressing the elegant ideas of electromagnetic theory. We've seen how scalar fields and vector fields can be encoded as types, and how these types make central ideas of EM theory clear. Aspects of vector calculus, like line and surface integrals, can be used to express the Maxwell equations. A Haskell version of the Biot-Savart law is arguably more understandable than its traditional mathematical form. Additional examples of electromagnetic theory in Haskell can be found in [8]. My anecdotal evidence that this is an effective way to teach the subject comes from students this year performing better on the *analytical* portions of exams than in prior years. Although the number of students (10) is small, I take this to mean that the process of programming in Haskell helped students get a better handle on the basic ideas I'm trying to teach. The author anticipates that dependently-typed functional languages could have even more to offer to physics and EM theory. For those who want to express the ideas of electromagnetic theory in beautiful code, functional languages are hard to beat.

References

[1] Harold Abelson, Gerald Jay Sussman & Julie Sussman (1996): *Structure and Interpretation of Computer Programs*, second edition. MIT Press.

[2] David J. Griffiths (2017): *Introduction to Electrodynamics*. Cambridge University Press, doi:10.1017/9781108333511.

[3] Seymour A. Papert (1993): *Mindstorms: Children, Computers, And Powerful Ideas*, 2 edition. Basic Books.

[4] Gerald Jay Sussman & Jack Wisdom (2001): *Structure and Interpretation of Classical Mechanics*. The MIT Press.

[5] Gerald Jay Sussman & Jack Wisdom (2013): *Functional Differential Geometry*. The MIT Press.

[6] Scott N. Walck (2014): *Learn Physics by Programming in Haskell*. In James Caldwell, Philip Hölzenspies & Peter Achten, editors: Proceedings 3rd International Workshop on *Trends in Functional Programming in Education*, Soesterberg, The Netherlands, 25th May 2014, *Electronic Proceedings in Theoretical Computer Science* 170, Open Publishing Association, pp. 67–77, doi:10.4204/EPTCS.170.5.

[7] Scott N. Walck (2016): *Learn Quantum Mechanics with Haskell*. In Johan Jeuring & Jay McCarthy, editors: Proceedings of the 4th and 5th International Workshop on *Trends in Functional Programming in Education*, Sophia-Antipolis, France and University of Maryland College Park, USA, 2nd June 2015 and 7th June 2016, *Electronic Proceedings in Theoretical Computer Science* 230, Open Publishing Association, pp. 31–46, doi:10.4204/EPTCS.230.3.

[8] Scott N. Walck (2023): *Learn Physics with Functional Programming: A Hands-on Guide to Exploring Physics with Haskell*. No Starch Press.

[9] Brent Yorgey (2011–2016): *The diagrams package*. `https://hackage.haskell.org/package/diagrams`.

Finite-State Automaton To/From Regular Expression Visualization

Marco T. Morazán and Tijana Minić
Seton Hall University
{morazanm|minictij}@shu.edu

Most Formal Languages and Automata Theory courses explore the duality between computation models to recognize words in a language and computation models to generate words in a language. For students unaccustomed to formal statements, these transformations are rarely intuitive. To assist students with such transformations, visualization tools can play a pivotal role. This article presents visualization tools developed for FSM–a domain-specific language for the Automata Theory classroom–to transform a finite state automaton to a regular expression and vice versa. Using these tools, the user may provide an arbitrary finite-state machine or an arbitrary regular expression and step forward and step backwards through a transformation. At each step, the visualization describes the step taken. The tools are outlined, their implementation is described, and they are compared with related work. In addition, empirical data collected from a control group is presented. The empirical data suggests that the tools are well-received, effective, and learning how to use them has a low extraneous cognitive load.

1 Introduction

Formal Languages and Automata Theory (FLAT) courses emphasize the equivalence of different computation models. For instance, the equivalence of deterministic finite-state machines (dfas) and nondeterministic finite-state machines (ndfas) is established by showing students how to transform an ndfa into a dfa. Such a transformation, although not trivial for a first-time FLAT student, is relatively intuitive: a machine (i.e., an ndfa) to recognize words in a language is transformed into a different machine (i.e., a dfa) to recognize words in the same language. In essence, an algorithm to determine language membership is transformed into another algorithm to determine language membership.

Less intuitive transformations are those from a model that recognizes words in a language to a model that generates words in a language and vice versa. That is, an algorithm to recognize words in a language is transformed into an algorithm to generate words in the same language and an algorithm to generate words in a language is transformed into an algorithm to recognize words in the same language. In such cases, the generated algorithm does not satisfy the same purpose. In a typical FLAT course, for instance, students learn how to transform a pushdown automaton into a context-free grammar and vice versa and learn how to transform a regular grammar into a finite-state automaton and vice versa.

Transformations from a generating algorithm to a recognizing algorithm and from a recognizing algorithm to a generating algorithm may be confusing for first-time FLAT students given that formal statements are rarely intuitive for them. Chief among these transformations is the one from an ndfa to a regular expression (regexp) and back. These transformations are important, because regexps are well-suited for humans to express patterns and finite-state automata are well-suited for program development [7]. For example, regexps are used in tools such as awk [25] and emacs [1] while finite-state automata are at the heart of algorithms for string searching [11] and lexical analysis [23]. To aid student understanding, visualization tools like JFLAP [26, 27] and OpenFlap [17] have been developed.

Stephen Chang (Ed.): Trends in Functional
Programming in Education (TFPIE)
EPTCS 405, 2024, pp. 36–55, doi:10.4204/EPTCS.405.3

It is usually assumed that visualizations are a powerful pedagogic tool in the classroom. They allow students to interact, to some degree or another, with their designs. This, however, may not be enough to have an effective teaching tool for several reasons. A visualization requires users to learn its interface and this can place an extraneous cognitive load on students [8, 29]. To be effective and reduce such a load, visualizations must provide representations that behave as the objects themselves [10]. This does not mean that a visualization tool cannot offer more advanced features. It means that it is important for there to be some easy-to-use features.

This article describes the visualization tools developed for `FSM` [21] to aid student understanding of the transformations from an `ndfa` to a `regexp` and from a `regexp` to an `ndfa`. `FSM` is a functional domain-specific language embedded in `Racket` [3] developed for the `FLAT` classroom to program state machines, grammars, and regular expressions [20]. The tools have multiple goals that include aiding in the understanding of the construction algorithms, reducing the extraneous cognitive load, and allowing students to examine construction steps interactively both forward and backwards. The article is organized as follows. Section 2 reviews and contrasts related work. Section 3 presents a brief introduction to the `FSM` syntax needed to navigate this article. Section 4 discusses the overall design idea behind the visualization strategies. Section 5 outlines the generation of visualization graphics. Section 6 presents empirical data collected, in preparation for classroom deployment, from a control group. Finally, Section 7 delivers concluding remarks and discusses directions for future work.

2 Related Work

2.1 Construction Algorithms

2.1.1 `regexp` to `ndfa`

Let Σ be the alphabet of a language. There are six regular expression varieties [28]:

1. R = a, where $a \in \Sigma$
2. R = ε, where ε denotes the empty word
3. R = $\emptyset$, denotes the empty language
4. R = $R_1 \cup R_2$, denotes the union of two regular expressions
5. R = $R_1 \circ R_2$, denotes the concatenation of two regular expressions
6. R = R_1^*, denotes zero or more concatenations of a regular expression

The creation of an `ndfa` for varieties 1–3 is straightforward. For varieties 1–2, each corresponding `ndfa` has a starting state and a final state. The transition between these consumes an alphabet element for the first variety and nothing (i.e., the empty string) for the second variety. The `ndfa` for the third variety only has a starting state and no final state. The transformation for varieties 4–6 hinges on closure properties for regular languages. That is, an `ndfa` is constructed using the algorithms developed as part of the constructive proofs establishing that the languages accepted by `ndfa`s are closed under union, concatenation, and Kleene star. For union and concatenation, `ndfa`s M_1 and M_2 are recursively constructed for R_1 and R_2. For union, a new starting and a new final state are created. The resulting `ndfa` nondeterministically moves from the new starting state to either M_1's or M_2's starting state and from each of M_1's and M_2's final states to the new final state. For concatenation, nondeterministic transitions are added from M_1's final states to M_2's starting state and the new machine's final states are M_2's final states. For Kleene star, an `ndfa`, M_1, is recursively constructed for `R1`. A new starting state, that is also a final state, is generated.

In addition, nondeterministic transitions from the new start state to M_1's start state and from M_1's final states to M_1's start state are generated. The reader may consult any introductory FLAT textbook for the formal details of these constructors (e.g., [9, 13, 15, 16, 20, 24, 28].

A transformation is commonly explained using a generalized nondeterministic finite-state automata (GNFA) [28]. A GNFA is similar to an ndfa, but its transitions are done on regular expressions. The initial GNFA has two states and a transition between them. The transition is on the regular expression that is being transformed. We have chosen this approach for our visualization tools, because at each step a single compound regular expression (i.e., union, concatenation, or Kleene star) may be decomposed to create the necessary new sub-GNFAs. The focus on a single edge facilitates the generation of an informative message that explains the step taken and, thus, reduces the extraneous cognitive load for students.

2.1.2 ndfa to regexp

FLAT textbooks usually outline the ndfa to regexp transformation using either an elegant set of recursive equations or a graph-based approach using the ndfa's transition diagram. The equation-based approach represents the language of the machine constructed as the union of a finite number of small languages [13]. By numbering the machine states $K=\{k_1, k_2, \ldots, k_n\}$, where k_1 is the starting state, the regular expression for all words that take the machine from state k_i to state k_j without traversing a state numbered m+1 or greater is denoted by R(i,j,m)[1]. Therefore, we have that the regular expression for the language of an ndfa, N, with n states is constructed as follows:

$$L(N) = \bigcup\{R(1,j,n) \mid k_j \in F\}, \text{ where F is N's set of final states}$$

That is, N's language contains all words that take the machine from the starting state to a final state by traversing any state. Assuming Δ is the given machine's set of transition rules, the regular expression is constructed using the following algorithm:

$$R(i,j,n) = \begin{cases} \{a|(k_i\ a\ k_j) \in \Delta\} & \text{if } n = 0 \wedge i \neq j \\ \{\varepsilon\}\bigcup\{a|(k_i\ a\ k_j) \in \Delta\} & \text{if } n = 0 \wedge i = j \\ R(i,j,n-1)\bigcup R(i,n,n-1)R(n,n,n-1)^{*}R(n,j,n-1) & \text{if } n \neq 0 \end{cases}$$

This recursive equation states that two cases are distinguished when there can be no intermediate states traversed (i.e., n=0). The first is when $k_i \neq k_j$. In this case, we have that all the singletons consumed by rules that directly transition from k_i to k_j are the needed regular expressions. The second is when $k_i = k_j$. In this case, ε is added to the set of symbols consumed on a self loop. If intermediate states may be traversed (i.e., n$\neq$0) then the needed regular expression is the union of two regular expressions. The first generates all words that take the machine from k_i to k_j without traversing a state numbered n or greater. The second concatenates three regular expressions: one that generates all words that take the machine from k_i to k_n without traversing a state greater than n-1, one that generates all words that take the machine from k_n to k_n an arbitrary number of times without traversing a state greater than n-1, and one that generates all words that take the machine from k_n to k_j without traversing a state greater than n-1.

The graph-based approach converts the ndfa, N, into a GNFA and then converts the GNFA to a regular expression [28]. The transformation may be visualized as performing surgery on a directed graph. A GNFA is built starting with N's transition diagram and adding a new start state, a new final state, and empty transitions from the new start state to N's start state and from N's final states to the new final state.

[1] An intermediate state is denoted as k_r, such that $0 \leq r \leq m$. The number of intermediate states is not relevant.

Figure 1 Initial step in the JFLAP visualization.

(a) GNFA for L=$\{ab^* \cup ba^*\}$.

(b) GNFA after one step.

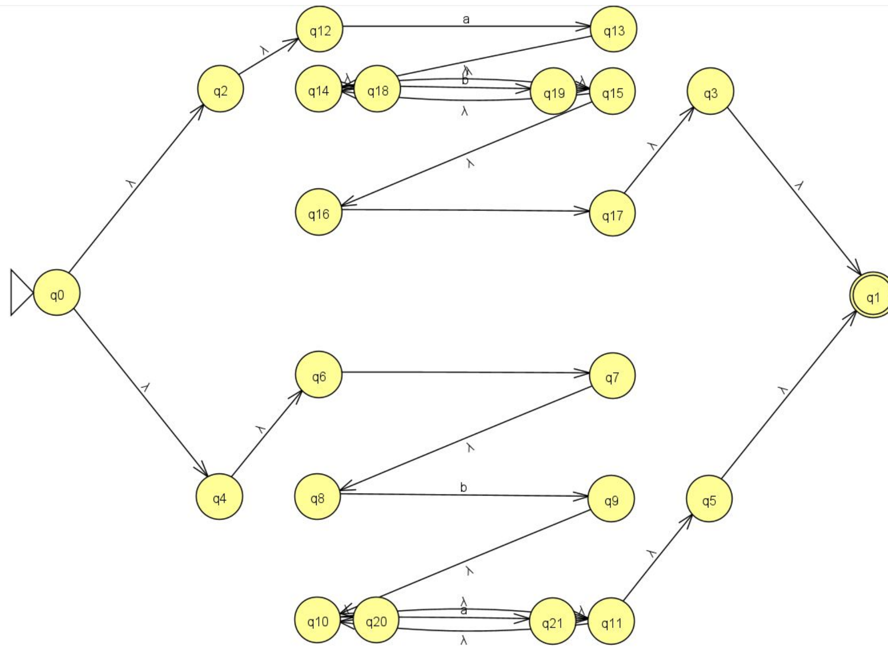
In addition, there is a single edge in each direction between any pair of states k_i and k_j. If there are one or more transitions from k_i to k_j then the label on the edge in the GNFA is a union-regexp containing a singleton-regexp for each of these transitions. Finally, if there are no edges from k_i to k_j then the label of the arrow is a null-regexp. Such added transitions do not change N's language because they represent hypothetical transitions and can never be used. The regular expression's computation proceeds by ripping out nodes piecemeal until only the new start state and the new final state remain. At this point, the regular expression on the only remaining edge is for the language of N. When a node k_r is ripped out, the edges into k_r, (k_i a k_r), and the edges out of k_r, (k_r b k_j) are replaced. If k_r does not have a self-loop then (k_i a k_r) and (k_r b k_j) are replaced with (k_i ab k_j). If there is a self-loop on k_r then (k_r b k_j), (k_r c k_r), and (k_r b k_j) are replaced by (k_i ac*b k_j). Finally, after ripping out k_r, multiple edges between nodes are replaced with a single edge that is labeled with the union of labels of the multiple edges.

FSM's visualization uses a graph-based approach, because it tends to be easier to understand by first-time FLAT students. The small local step approach of ripping out one node at a time is more palatable than, for example, computing R(i,j,n) which requires a more global view of the transition relation. The algorithm used, like the algorithm described by Sipser [28], builds a GNFA by adding new a new start state, a new final state, and the corresponding empty transitions. In contrast, however, the addition of edges with a null-regexp label is suppressed. Such edges serve no real purpose and, for visualization purposes, clutter the graphics produced. Instead of adding such edges, when a node is ripped out its predecessors and its successors are computed to properly substitute the edges into and out of the ripped out node. In this manner, the graphs produced are easier to read by students and lower the extraneous cognitive load.

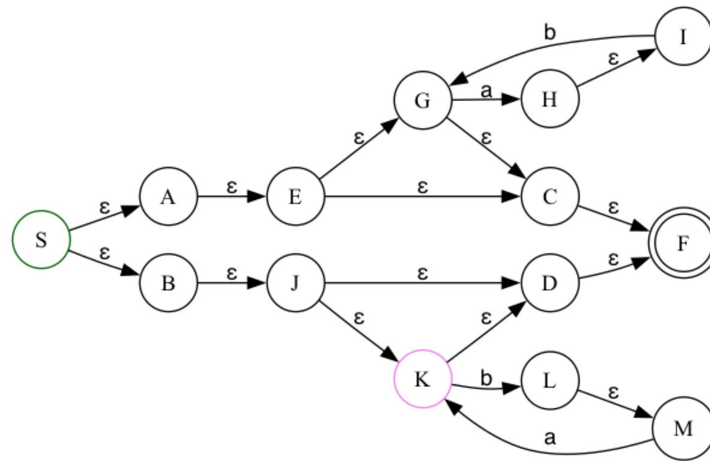
2.2 Visualization

JFLAP is a visualization tool that supports the ndfa to regexp transformation and vice versa. To transform from an ndfa to a regexp, the user must manually construct the ndfa. This includes graphically drawing nodes and edges, marking the starting and final states, and laying out the transition diagram in an appealing manner. To transform from a regexp to an ndfa, the user must use a concrete grammar to write the regular expression. This grammar uses + for union, * for Kleene star, ! for the empty word, and parentheses to define the order of operations. For neither conversion is the user given the ability to step back through the computation to review previous steps.

Figure 2 Resulting transition diagrams for L=$\{ab^* \cup ba^*\}$.



(a) JFLAP transition diagram.



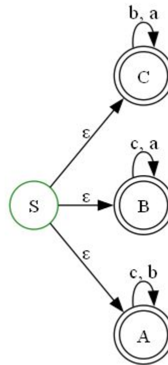
(b) FSM transition diagram.

2.2.1 regexp to ndfa Visualization

The conversion is done from a GNFA to regexp. At each step, a decomposable regular expression is transformed using ndfa closure properties over union, concatenation, and Kleene star. Such a step may be manually done by the user or may be done automatically by pressing a Do Step button. To illustrate how a step is done, consider the GNFA in Figure 1a for L=$\{ab^* \cup ba^*\}$[2]. The result of performing the first transformation step results in the GNFA displayed in Figure 1b. The edge from q0 to q1 is substituted

[2]The nodes have been manually rearranged to make the illustrations easy to read.

Figure 3 An `ndfa` for the language L={w | w is missing at least one in {a b c}}.



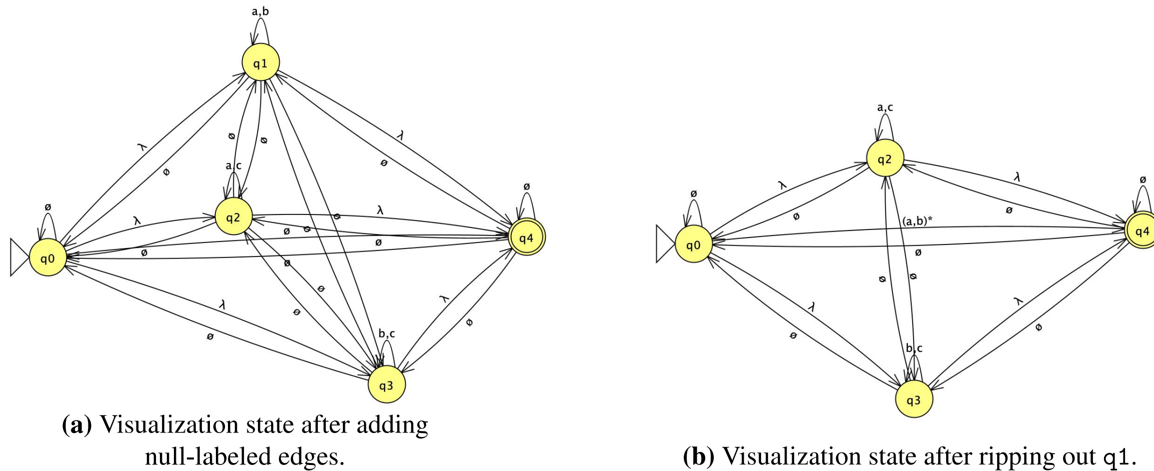
with a `GNFA` that starts at `q0`, nondeterministically transitions to a (sub-)`GNFA` for one of the union's branches, and from both (sub-)`GNFA`s nondeterministically transitions to `q1`. The conversion process may continue piecemeal or may be completed in one step by pressing a `Do All` button that results, without manually rearranging nodes to improve readability, in the `ndfa` displayed in Figure 2a. As the reader can appreciate, nodes are haphazardly placed and some edges are impossible to read, thus, requiring the user to rearrange nodes to make the transition diagram readable.

`FSM`'s visualization also uses a graph-based approach to generate the `ndfa`. Like `JFLAP`, closure properties of regular languages over union, concatenation, and Kleene star are used to transform decomposable `regexp`s. In contrast, however, a primary goal of the `FSM` visualization is to reduce the extraneous cognitive load. To this end, the tool always selects the next decomposable `regexp` to transform and allows the user to step back in the computation to review transformation steps. In addition, an informative message is displayed highlighting the edge that is transformed. In this manner, the user does not have to wonder what occurred when a step is not clear to them. In further contrast with `JFLAP`, every transition diagram is rendered using `Graphviz` [4, 6]. Thus, nodes and edges are rendered in an appealing manner that makes reading the rendered transition diagram easier. For instance, compare the transition diagram layout obtained using `FSM`'s visualization displayed in Figure 2b with its counterpart obtained using `JFLAP` in Figure 2a.

2.2.2 `ndfa` to `regexp`

`JFLAP`'s `ndfa` to `regexp` transformation visualization follows the graph-based approach described above. The user manually performs each step of the transformation by following provided instructions. The instructions have the user add a new dead state, create a `GNFA` by combining edges with multiple labels into a union `regexp`, add null-labeled directed edges between states that do not have an edge between them, and, finally, rip out nodes. The user has the option to complete each of these steps manually or automatically. To illustrate the conversion, consider transforming the `ndfa` displayed in Figure 3. Figure 4a displays the visualization's state after adding a new final state and the necessary null transitions between states (nodes have been moved to improve readability). Despite moving nodes to improve readability, we can observe that the visualization's state is hard to read at best. The primary problem is that the visualization is cluttered with null-labeled edges that result in edge overlapping. Figure 4b displays the visualization state after ripping out the first node (`q1` in this example). We can once again observe that the visualization state is hard to read. In addition, it is difficult to visually discern the effect of ripping

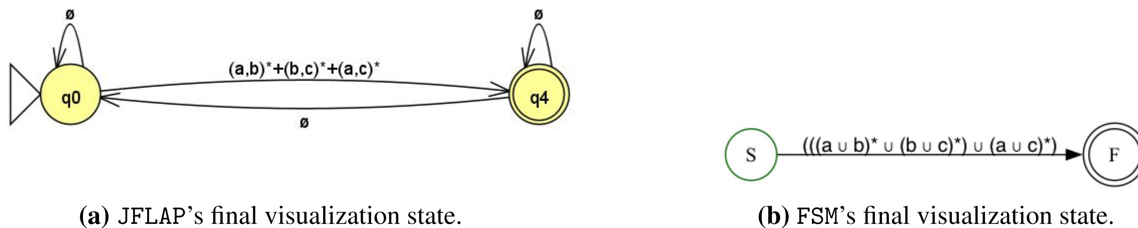
Figure 4 `JFLAP` visualization states.



(a) Visualization state after adding null-labeled edges.

(b) Visualization state after ripping out `q1`.

Figure 5 `JFLAP`'s and `FSM`'s final visualization state.



(a) `JFLAP`'s final visualization state.

(b) `FSM`'s final visualization state.

out a node. Finally, Figure 5a displays `JFLAP`'s final visualization state. In the final state, it becomes easy to discern the resulting regular expression despite the (useless) null-labeled transitions.

As in `JFLAP`, `FSM`'s visualization also follows the graph-based transformation approach. In contrast, however, the user is much less burdened. The user does not have to manually add a final state, create a `GNFA`, add null-labeled directed edges, nor choose the next node to rip out. All this is automatically done or omitted as the user steps through the visualization. Thus, reducing the extraneous cognitive load. The user only needs to step forward and backward between visualization states using the arrow keys. The ability to step backwards in the transformation allows users to visually observe how edges are combined when a node is ripped out. Finally, every transition diagram is rendered using `Graphviz` [4, 6] to provide an appealing layout. For instance, for the `ndfa` displayed in Figure 3, the `FSM` visualization's final state is rendered as displayed in Figure 5b. The reader can appreciate that the graphic is more appealing than the graphic produced by `JFLAP`.

3 A Brief Introduction to `FSM`

`FSM` is a domain specific language, embedded in `Racket`, for the `FLAT` classroom. In `FSM`, state machines, grammars, and regular expressions are first-class. Nondeterminism is a built-in language feature that programmers may use as they use their favorite features in any programming language. The `FSM` types relevant for this article are regular expressions and finite-state machines.

Figure 6 The FSM regular expression for $L=\{ab^* \cup ba^*\}$.

```
#lang fsm

(define a (singleton-regexp "a"))     (define b (singleton-regexp "b"))

(define a* (kleenestar-regexp a))     (define b* (kleenestar-regexp b))

(define ab* (concat-regexp a b*))     (define ba* (concat-regexp b a*))

;; L= ab* U ba*
(define ab*Uba* (union-regexp (concat-regexp a b*) (concat-regexp b a*)))

;; word → Boolean
;; Purpose: Determine if the given word is in ab* U ba*
(define (in-ab*Uba*? w)
  (or (and (eq? (first w) 'a) (andmap (λ (s) (eq? s 'b)) (rest w)))
      (and (eq? (first w) 'b) (andmap (λ (s) (eq? s 'a)) (rest w)))))

(check-pred in-ab*Uba*? (gen-regexp-word ab*Uba*))
(check-pred in-ab*Uba*? (gen-regexp-word ab*Uba*))
(check-pred in-ab*Uba*? (gen-regexp-word ab*Uba*))
```

3.1 Regular Expressions

The constructors for a regular expression, over an alphabet Σ, are:

1. (null-regexp)
2. (empty-regexp)
3. (singleton-regexp "a"), where a∈Σ
4. (union-regexp r1 r2), where r1 and r2 are regular expressions
5. (concat-regexp r1 r2), where r1 and r2 are regular expressions
6. (kleenestar-regexp r1), where r is a regular expression

The FSM selector functions for regular expressions are:

singleton-regexp-a: Extracts the embedded string
kleenestar-regexp-r1: Extracts the embedded regular expression
union-regexp-r1: Extracts the first embedded regular expression
union-regexp-r2: Extracts the second embedded regular expression
concat-regexp-r1: Extracts the first embedded regular expression
concat-regexp-r2: Extracts the second embedded regular expression

The following predicates are defined to distinguish among the regular expression subtypes:

```
empty-regexp?     singleton-regexp?     kleenestar-regexp?
union-regexp?     concat-regexp?        null-regexp?
```

Each consumes a value of any type and returns a Boolean. Finally, the observer, `gen-regexp-word`, takes as input a regular expression and returns a word in the language of the given regular expression. This observer nondeterministically decides how many repetitions of a `kleenestar-regexp` to generate and nondeterministically decides which branch of a `union-regexp` to use in generation.

As a programming example, consider the `FSM` regular expression displayed in Figure 6 for L={ab* ∪ ba*}. The code is made readable by independently defining each needed sub-`regexp`. The reader can appreciate that this makes the implementation accessible for almost any student. The unit tests use the auxiliary predicate, `in-ab*Uba*?`, to determine if a generated word is in `L`. The tests all look the same, but they are not (in all likelihood) given that each word generation, as described above, nondeterministically decides the number of repetitions for a Kleene star and the branch of the union used to generate a word.

3.2 Finite-State Automatons

The `FSM` machine constructors of interest for this article are those for finite-state automata:

```
make-dfa:  K Σ s F δ → dfa      make-ndfa: K Σ s F δ → ndfa
```

`K` is a list of states, Σ is a list of alphabet symbols, s∈K is the starting state, F⊆K is a list of final states, and δ is a transition relation (that must be a function for a `dfa`). A transition relation is represented as a list of transition rules. A `dfa` transition rule is a triple, `(K Σ K)`, containing a source state, the element to read, and a destination state. For an `ndfa` transition, the element to read may be `EMP` (i.e., nothing is read).

The observers are:

```
(sm-states M) (sm-sigma M)   (sm-start M) (sm-finals M) (sm-rules M)
(sm-type M)   (sm-apply M w) (sm-showtransitions M w)
```

The first 5 observers extract a component from the given state machine, `sm-type` returns the given state machine's type, `sm-apply` applies the given machine to the given word and returns `'accept` or `'reject`, and `sm-showtransitions` returns a trace of the configurations traversed when applying the given machine to the given word ending with the result. A trace is only returned, however, if the machine is a `dfa` or if the word is accepted by an `ndfa`.

Finally, `FSM` provides machine rendering and machine execution visualization. The visualization primitives are:

```
(sm-graph M)     (sm-visualize M [(s p)*])
```

The first returns an image for the given machine's transition diagram. The second launches the `FSM` visualization tool. The optional two-lists, `(s p)`, contain a state of the given machine and an invariant predicate for the state. Machine execution may always be visualized if the machine is a `dfa`. Similarly to `sm-showtransitions`, `ndfa` machine execution may only be visualized if the given word is in the machine's language. For further details on machine execution visualization in `FSM`, the reader is referred to a previous publication [22].

To illustrate programming finite-state machines in `FSM`, consider the `ndfa` to decide L={ab* ∪ ba*} displayed in Figure 7. At the beginning, the machine nondeterministically decides if the given word is in ab^* or in ba^* and transitions, respectively, to `A` or `D`. The unit tests illustrate words that are and that are not in `L`. Observe that the programmer only specifies nondeterministic behavior (the transitions out of `S`) and is not burdened with implementing nondeterministic behavior.

Figure 7 The FSM ndfa for L={ab* ∪ ba*}.

```
#lang fsm

;; L= ab* U ba*
(define ab*Uba*-ndfa (make-ndfa '(S A B D E)
                                '(a b)
                                'S
                                '(B C E F)
                                `((S ,EMP A) (S ,EMP D)
                                  (A a B) (B b B)
                                  (D b E) (E a E))))

(check-equal? (sm-apply ab*Uba*-ndfa '(b b))   'reject)
(check-equal? (sm-apply ab*Uba*-ndfa '(a a b)) 'reject)
(check-equal? (sm-apply ab*Uba*-ndfa '(a))     'accept)
(check-equal? (sm-apply ab*Uba*-ndfa '(b))     'accept)
(check-equal? (sm-apply ab*Uba*-ndfa '(a b b)) 'accept)
(check-equal? (sm-apply ab*Uba*-ndfa '(b a))   'accept)
```

4 Overall Visualization Design

To reduce the extraneous cognitive load, FSM's visualizations generate and collect images for each transformation step. As part of each image, there is a brief informative message that explains the step taken. The user only needs to use the arrow keys to step through the transformation. The use of these keys is specified as follows:

→	Move to next visualization step	←	Move to previous visualization step
↓	Move to visualization's end	↑	Move to visualization's start

The visualization always displays instructions for the use of the arrow keys.

The images are stored in a structure, viz-state, that is defined as follows:

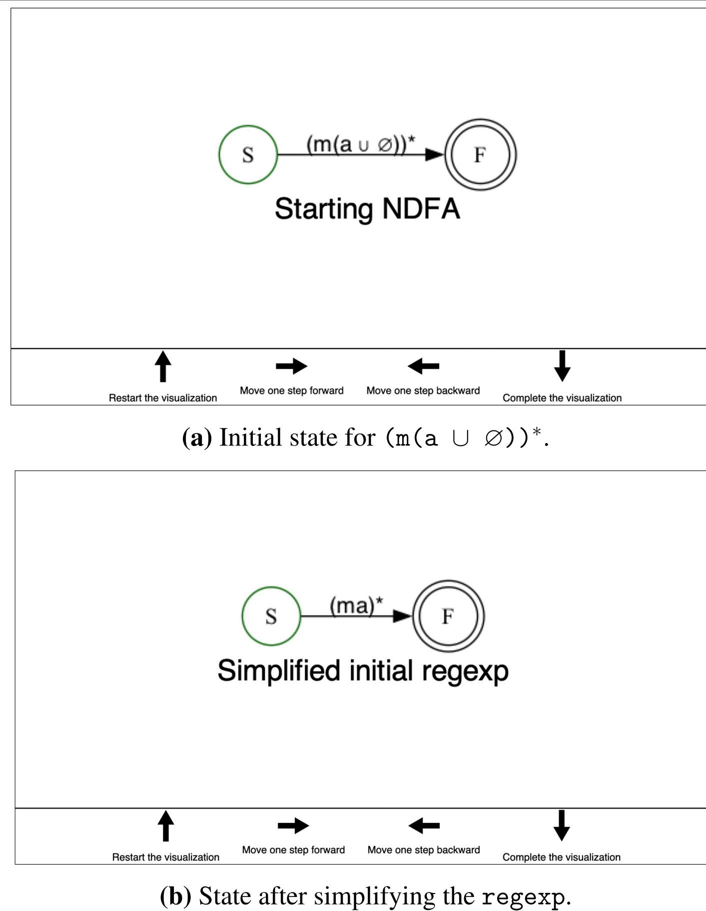
```
;; A structure, (viz-state (listof images) (listof images)),
;; containing the processed and unprocessed images.
(struct viz-state (pimgs upimgs))
```

The first list, pimgs, denotes the images previously displayed. The second list, upimgs, denotes the images to be displayed. The first image in upimgs is the currently displayed image. Initially, all images are in upimgs. Using the right arrow moves the first image from upimgs to pimgs and using the left arrow does the opposite. Using the down arrow moves all images from upimgs, except the last one, to pimgs. Using the up arrow moves all images to upimgs. Every time a step forward or backwards is taken, an informative message is placed at the bottom of each graphic along with arrow-use instructions. When appropriate, color is used to highlight the changes in the transformation.

4.1 Illustrative Example: regexp to ndfa

For illustrative purposes, consider the transforming $(\texttt{m}(\texttt{a} \cup \varnothing))^*$ into an ndfa. The visualization's initial GNFA is displayed in Figure 8a. It contains a single transition from the starting state to the final

Figure 8 First visualization states in the `regexp` to `ndfa` transformation.



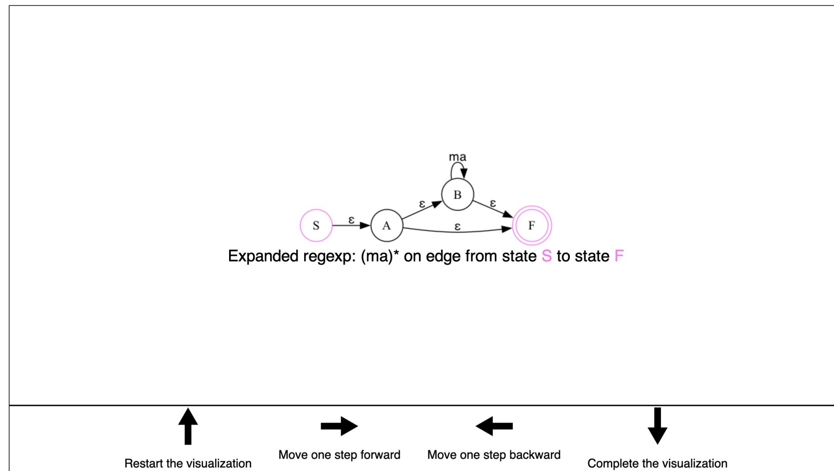
(a) Initial state for $(\mathtt{m}(\mathtt{a} \cup \varnothing))^*$.

(b) State after simplifying the `regexp`.

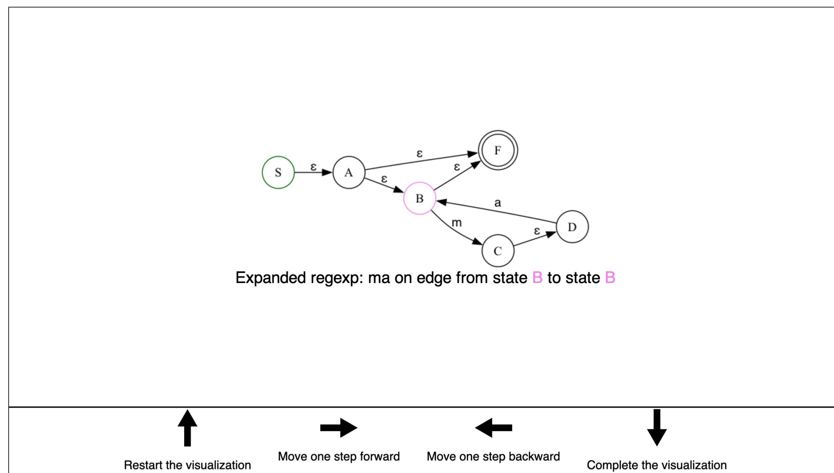
state labeled with the `regexp` to transform. The message indicates that it is the starting (approximation of the) `ndfa`. The first step simplifies the given regular expression to $(\mathtt{ma})^*$. The visualization's state after this step is displayed in Figure 8b. The message informs the user that the initial regular expression has been simplified. Technically, this step is not necessary but is useful to make the visualization more comprehensible for students that tend to write overly complex `regexps`. Next, the Kleene star `regexp` that takes the machine from state `S` to state `F` is transformed. The visualization's state after this step is displayed in Figure 9a. Observe that the message indicates the regular expression expanded, and the source and destination states. In both the message and in the graphic these states are highlighted in violet. The final step in the transformation expands `ma`. Given that this regular expression is on `B`'s self-transition, the source and destination states are the same. The state of the visualization after this expansion is displayed in Figure 9b. Observe that the message indicates the `regexp` expanded and highlights in violet a single state.

At any point in the transformation, the user may move backwards in the transformation to examine before and after visualization states. This feature, along with the provided messages, allows the user to examine closely how the transformation is advanced by each step.

Figure 9 Final visualization states in the `regexp` to `ndfa` transformation.



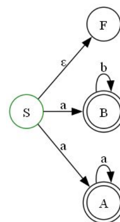
(a) Visualization state after expanding `(ma)*`.



(b) Visualization state after expanding `ma`.

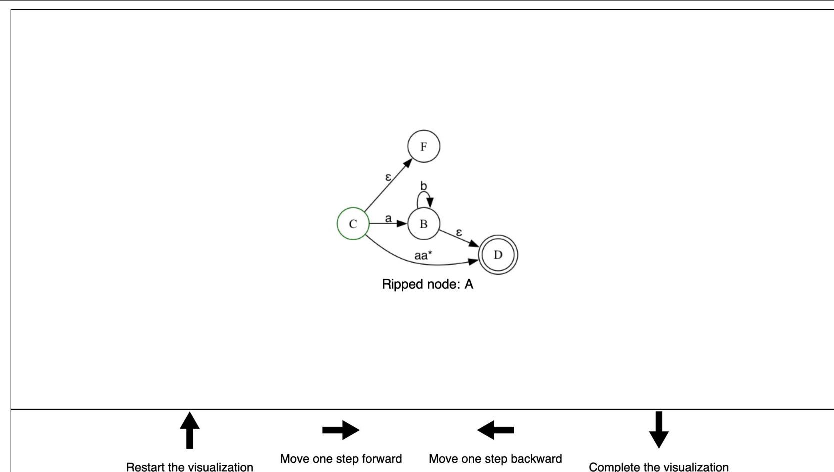
4.2 Illustrative Example: `ndfa` to `regexp`

For illustrative purposes, consider transforming the following `ndfa`:

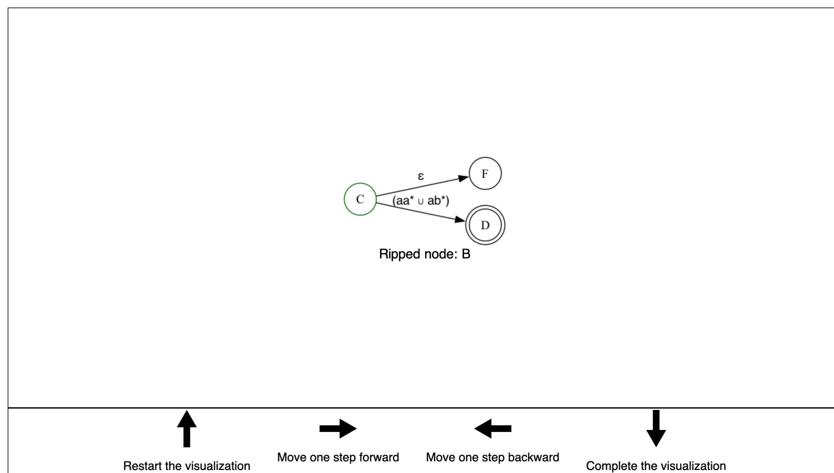


The programmer has, unnecessarily, included a nonfinal state, `F`, that is only reachable by an empty transition from the starting state and that does not have any outgoing transitions. The visualization steps rip out a node one at a time. Figure 10a displays the visualization state after ripping out `S` and `A`. Observe that there is a transition on `aa*` from `C` to `D` resulting from ripping out the two nodes. Ripping out `B`

Figure 10 Node-ripping visualization steps in the `ndfa` to `regexp` transformation.



(a) Visualization state after ripping out `S` and `A`.



(b) Visualization state after ripping out `B`.

means a new transition is needed from `C`, the only predecessor, to, `D`, the only successor. This results in two edges between `C` and `D` and, thus, they are consolidated using a union regular expression resulting in the visualization state displayed in Figure 10b. Finally, ripping out `F` has no effect on the edge from `C` and `D`, which is labeled with the resulting regular expression.

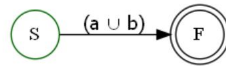
At any point in the transformation, the user may move backwards to examine before and after visualization states. Thus, allowing the student to closely examine how nodes are ripped out and new transitions are created.

5 Implementation

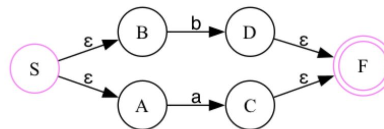
5.1 Constructing the Graphics for the `regexp` to `ndfa` Transformation

During the transformation, there exists a GNFA whose transitions are labeled with arbitrary regexps. The goal is to transform the GNFA so that its transitions are only labeled with singleton and empty regexps. At each step, a transition labeled with a union, a concatenation, or a Kleene star regexp is chosen to be transformed. These transformations are based on well-known constructors for closure properties of regular languages [13, 20, 24, 28]. When a regexp is transformed, the chosen transition is removed from the GNFA and new states and edges are added. Graphviz is used to generate a new graphic.

A union regexp, (union-regexp r_1 r_2), labeling the transition between two states S and F is transformed by creating four fresh states: say, A, B, C, and D. Each branch of the union exclusively uses two of these states and they are connected by a transition labeled with the corresponding regular expression for the branch. For instance, A, and C are connected using r_1 and B, D are connected using r_2. S is connected to A and B by empty transitions. C and D are connected to F by empty transitions. Visually, transforming:

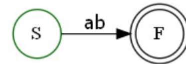


results in:

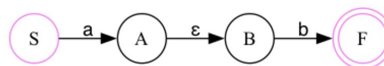


Finally, S and F are highlighted in violet indicating the head and the tail of the replaced edge.

A concatenation regexp, (concat-regexp r_1 r_2), labeling the transition between two states S and F is transformed by creating two fresh states: say, A and B. A transition from S to A labeled with r_1, a transition from A to B labeled with an empty regexp, and a transition from B to F labeled with r_2 are added to the GNFA. Visually, transforming:

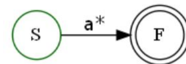


results in:

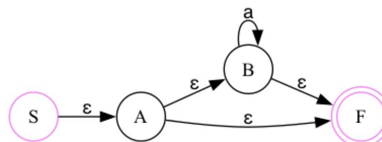


Finally, S and F are highlighted in violet indicating the head and the tail of the replaced edge.

A Kleene star regexp, (kleenestar-regexp r_1), labeling the transition between two states S and F is transformed by creating two fresh states: say, A and B. S is connected to A, A is connected to B, A is connected to F, and B is connected to F by transitions labeled with an empty regexp. Finally, there is a loop transition on B labeled with r_1. Visually, transforming:



results in:

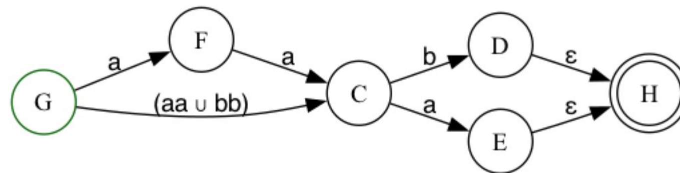


Finally, S and F are highlighted in violet indicating the head and the tail of the replaced edge.

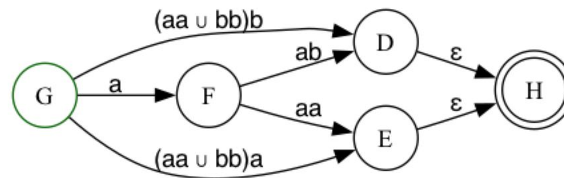
5.2 Constructing the Graphics for the `ndfa` to `regexp` Transformation

The bulk of the graphics are created by ripping out nodes. Ripping out a node `A` requires the removal of transitions into and out of `A` and the generation of new transitions connecting each predecessor of `A` with each successor of `A`. There are two cases that need to be distinguished: either `A` has or does not have a loop transition on it.

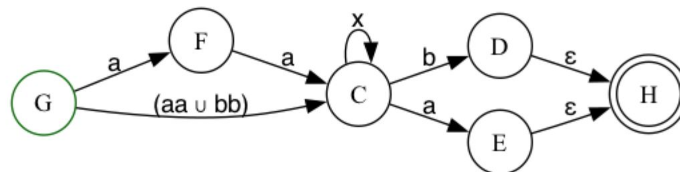
If `A` does not have a loop on it then each predecessor of `A` is connected to each successor of `A` by a transition labeled with a concatenation `regexp` that contains the regular expression from the predecessor to `A` and the regular expression from `A` to the successor. For instance, if (M r_1 A) and (A r_2 N) are transitions in the current `GNFA` then these two transitions are removed and substituted with (M r_1r_2 N). Visually, if the current `GNFA` is:



Ripping out `C` means that `G` and `F` must be connected to `E` and `D`. The new edges generated are labeled with the concatenation of each edge into `C` and each edge out of `C`. The resulting `GNFA` is:



If `A` has a loop on itself then each predecessor of `A` is connected to each successor of `A` by a transition labeled with the concatenation of the regular expression from the predecessor to `A`, a Kleene star regular expression for the loop's regular expression, and the regular expression to the successor of `A`. For instance, if (M r_1 A), (A r_2 A) and (A r_3 N) are transitions in the current `GNFA` then, when `A` is ripped out, these three transitions are removed and substituted with (M $r_1r_2^*r_3$ N). Visually, if the current `GNFA` is:



Ripping out `C` means that `F` and `G` must have transitions to `E` and `D`. The new edges generated are labeled with the concatenation of each edge into `C`, a Kleene star regular expression containing the label on `C`'s loop, and the edges out of `C`. The resulting `GNFA` is:

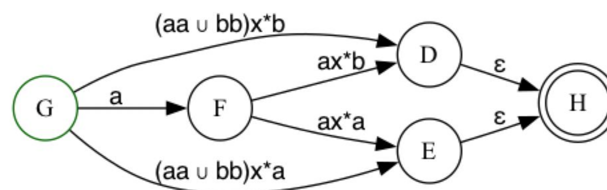
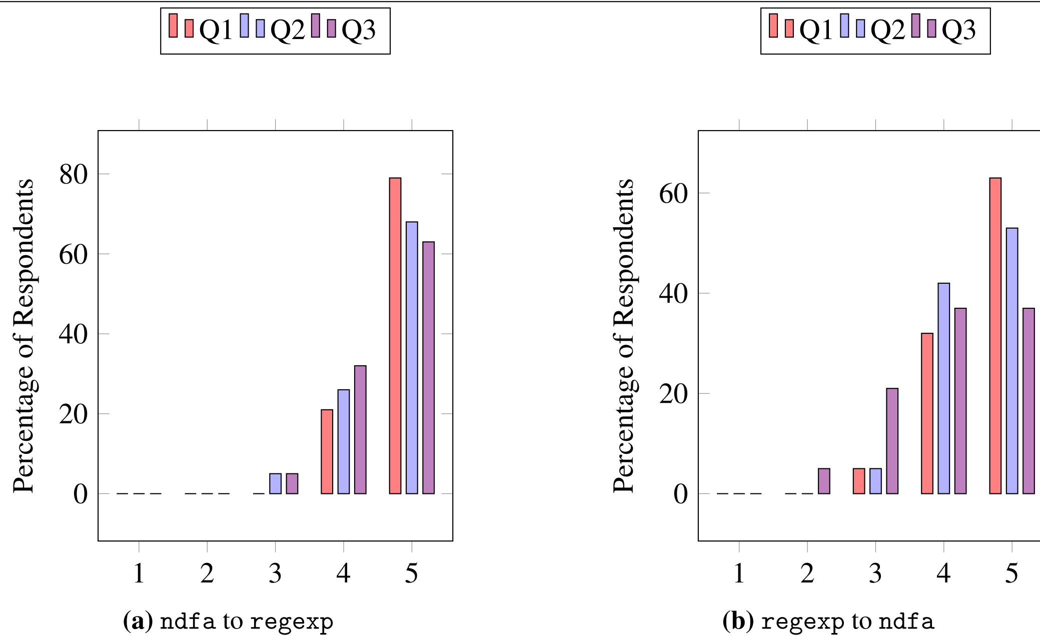


Figure 11 Control group response distribution.



6 Empirical Data

To initially assess the usefulness of our new teaching tools, before deploying them in a classroom setting, we collected empirical data from a focus group using a voluntary anonymous survey[3]. Nineteen students volunteered to participate (5 Seton Hall undergraduates who have not yet taken a `FLAT` class; 13 Instituto Universitário de Lisboa undergraduates currently taking a `FLAT` class, and 1 graduate student that works as a `FLAT` teaching assistant at Instituto Universitário de Lisboa)[4]. They were all introduced to the `ndfa` to `regexp` and to the `regexp` to `ndfa` transformations using the `FSM` tools described in this article. Prior to learning about the transformations, students got a brief introduction to `FSM` focusing on programming `ndfa`s and `regexp`s. After learning about each of the transformations and using the visualization tools, the students took a survey with the following questions about the `ndfa` to `regexp` visualization tool:

> Q1: Overall, how useful is the visualization to understand the ndfa to regexp transformation?
> Q2: How difficult is it to use the visualization?
> Q3: How difficult is it to understand a visualized transformation?

Respondents answered using a Likert scale [14]. Question 1 uses the scale from [1] Not at all useful to [5] Extremely useful. Questions 2 and 3 use the scale from [1] Extremely difficult to [5] Extremely easy. The distribution of responses is displayed in Figure 11a.

Responses to `Q1` indicate that all respondents feel that the visualization tool is useful to understand the transformation from `ndfa` to `regexp` (responses 4 and 5). These results were not anticipated given that most respondents have little to no experience with formal languages and automata theory. This suggests that the visualization is useful even for `FLAT` beginners.

[3]None of the volunteers received any benefits for their participation.

[4]Only the 5 volunteers from Seton Hall University are familiar with `Racket`-like languages (specifically, the `Racket` student languages used in [2, 18, 19])

Responses to Q2 indicate that most respondents, 94%, feel strongly that the visualization tool is easy to use (responses 4 and 5). This suggests that the efforts made to reduce the extraneous cognitive load associated with learning how to use the visualization are successful.

Responses for Q3 indicate that most respondents, 95%, feel strongly that the visualized transformation is easy to understand. This is also an unexpected result given that most respondents were not familiar with the transformation algorithms. It suggests that the size of each step in the visualization makes the transformation accessible to novices.

The second part of the survey addressed the regexp to ndfa transformation. The survey included the following questions:

Q1: Overall, how useful is the visualization to understand the regexp to ndfa transformation?
Q2: How difficult is it to use the visualization?
Q3: How difficult is it to understand a visualized transformation?

These questions are also answered using a Likert scale [14]. Question 1 uses the scale from [1] Not at all useful to [5] Extremely useful. Questions 2 and 3 use the scale from [1] Extremely difficult to [5] Extremely easy. The distribution of responses is displayed in Figure 11b.

For Q1, we observe that respondents feel strongly, with 95% answering 4 or 5, that the visualization tool is useful to understand the transformation from regexp to ndfa. These results are unexpected as most respondents, as observed earlier, have no prior experience with formal languages and automata theory. Along with the results obtained for Q1 for the previous transformation above, this suggests that providing a visual trace of construction algorithms benefits students at all levels of experience.

For Q2, we observe that most respondents, 95%, feel strongly that the visualization is easy to use. This suggest that our efforts to keep the extraneous cognitive load low are successful. We attribute this to the easy-to-use arrow-key interface and the informative messages at each step.

For Q3, we observe that a majority of respondents, 74%, feel that the transformation is easy to understand (answers 4 and 5). A significant minority of respondents, 21%, felt less strongly (response 3). Such a distribution is expected among students beginning in FLAT given that, to fully understand this transformation, the respondents need to be familiar with closure properties for regular languages and the corresponding construction algorithms. Nonetheless, these results are very encouraging given that even the novices felt they understood the transformation.

In addition, the respondents were asked qualitative questions. The following responses were obtained when respondents were asked about their favorite characteristics of the visualization tools:

```
"Being able to cycle step-by-step through each step in the
 visualization is super useful. I'm thinking of how my students
 might not understand a particular step, and I can just cycle
 back and forth as much as I need :) I also really liked the
 messages that explain what was done in each step, feel like
 they really help to keep track of what's going on!"

"It makes it much easier to juggle all the different states in
 my head."

"I like the purple highlight coloring of the node that is to be
 broken down."

"They are pretty straightforward and easy to understand.
```

```
 The colors are nice and easy to follow too."

"Really easy to understand what's going on. I will, for sure,
 use it for studying."
```

This feedback suggests that, due to the perceived clarity and the readability, students in a course setting will welcome and use the visualization tools.

The following responses were given when asked what they liked the least about the visualization tools:

```
"The long I-xyzw... state names can make the visualization
somewhat overwhelming in my opinion."

"Maybe the name of the new states should have a better
 naming scheme instead of random names"

"The movement of nodes instead of static points and growing frame."
```

The first two comments refer to the prior names randomly generated for new states. The prior names included a random 6-digit natural number (e.g., `I-872431`). In light of the above feedback and prior to publication, random state-name generation has been updated to only include, if necessary, a random number. The new generation technique produces the shortest possible state-name not in use in the construction of an `ndfa`. The use of this new random state-name generation is reflected in the previous sections of this article (i.e., a random state-name with a 6-digit natural number is not generated for any of the examples used).

The second concern refers to the placement of nodes in the generated graphs. Given that drawing graphs is complex, the main impetus for current research on computer-aided graph drawing is to facilitate the visual analysis of various kinds of complex networked or connected systems [12]. Several graph drawing libraries have been built and successfully deployed. Among the most widely used is `Graphviz` [5] and `FSM` uses `Graphviz` to generate its diagrams. `Graphviz`, however, provides no control over node placement and we must accept state movement as diagrams grow.

7 Concluding Remarks

This article presents novel `FSM` visualization tools for an `ndfa` to `regexp` transformation and for a `regexp` to `ndfa` transformation. The visualizations simultaneously render the transition diagram images and display informative messages to assist the user navigate the transformation. It improves the previous approaches by rendering transition diagrams in an appealing manner. In addition, the `regexp` to `ndfa` visualization tool has appropriate state color coding to clearly illustrate which edge has been expanded. The `FSM` visualization tools, unlike any other visualization tools for these transformations, can advance both forwards and backwards. Finally, both transformations can be completed in a single step, or restarted in a single click from any point in the computation, without preventing the user from moving the simulation forwards and backwards. All these advancements are done by clicking the arrow keys. Thus, helping lower the extraneous cognitive load associated with learning how to use the tools.

Future work includes using the described tools in a classroom setting and measuring student impressions. We envision using the visualizations to help students understand formal statements. That is,

the plan is to introduce students to the transformation algorithms using formal notation (given that it is important for students to understand formal statements) and, in tandem, to use the visualization tools to help students understand the formal notation so that they can implement the algorithms in `FSM`. Future work also includes developing visualization tools for construction algorithms based on closure properties for regular languages including union, concatenation, Kleene star, complement, and intersection. In addition, we are expanding the reach of our visualization tools into derivations for regular, context-free, and context-sensitive grammars. The goal is to assist students understand why a word is a member of a language through the creation of parse trees.

Acknowledgements.

The authors thank Filipe Alexandre Azinhais dos Santos and Alfonso Manuel Barral Caniço from Instituto Universitário de Lisboa for inviting us to their classroom to conduct our control group study. In addition, the authors thank Oliwia Kempinski, Andrés Maldonado, Josie Des Rosiers, and Shamil Dzhatdoyev for their feedback on previous versions of this manuscript.

References

[1] Richard Stallman et al. (2023): *GNU Emacs Manual*, version 29.1 edition. Free Software Foundation, Inc. Last accessed: November 2023.

[2] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2018): *How to Design Programs: An Introduction to Programming and Computing*, Second edition. MIT Press, Cambridge, MA, USA.

[3] Matthew Flatt, Robert Bruce Findler & PLT: *The Racket Guide*. Available at `https://docs.racket-lang.org/guide/`. Last accessed 2023-07-07.

[4] Emden R. Gansner & Stephen C. North (2000): *An Open Graph Visualization System and Its Applications to Software Engineering*. Softw. Pract. Exper. 30(11), p. 1203–1233, doi:10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

[5] Emden R. Gansner & Stephen C. North (2000): *An Open Graph Visualization System and Its Applications to Software Engineering*. Softw. Pract. Exper. 30(11), p. 1203–1233, doi:10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

[6] E.R. Gansner, E. Koutsofios, S.C. North & K.-P. Vo (1993): *A technique for drawing directed graphs*. IEEE Transactions on Software Engineering 19(3), pp. 214–230, doi:10.1109/32.221135.

[7] Hermann Gruber & Markus Holzer (2014): *From Finite Automata to Regular Expressions and Back-A Summary on Descriptional Complexity*. In Zoltán Ésik & Zoltán Fülöp, editors: Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014, EPTCS 151, pp. 25–48, doi:10.4204/EPTCS.151.2.

[8] Mary Hegarty (2004): *Dynamic Visualizations and Learning: Getting to the Difficult Questions*. Learning and Instruction 14(3), pp. 343–351, doi:10.1016/j.learninstruc.2004.06.007. Dynamic Visualisations and Learning.

[9] John E. Hopcroft, Rajeev Motwani & Jeffrey D. Ullman (2006): *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA.

[10] Edwin L. Hutchins, James D. Hollan & Donald A. Norman (1985): *Direct Manipulation Interfaces*. Hum.-Comput. Interact. 1(4), p. 311–338, doi:10.1207/s15327051hci0104_2.

[11] Donald E. Knuth, James H. Morris, Jr. & Vaughan R. Pratt (1977): *Fast Pattern Matching in Strings*. SIAM Journal on Computing 6(2), pp. 323–350, doi:10.1137/0206024.

[12] E. Kruja, J. Marks, A. Blair & R.C. Waters (2001): *A Short Note on the History of Graph Drawing*. In P. Mutzel, M. Junger & S. Leipert, editors: *International Symposium on Graph Drawing (GD)*, Lecture Notes in Computer Science, Springer, pp. 272–286, doi:10.1007/3-540-45848-4_22. Available at `https://www.merl.com/publications/TR2001-49`.

[13] Harry R. Lewis & Christos H. Papadimitriou (1997): *Elements of the Theory of Computation*, 2nd edition. Prentice Hall PTR, Upper Saddle River, NJ, USA, doi:10.1145/300307.1040360.

[14] Rensis Likert (1932): *A Technique for the Measurement of Attitudes*. *Archives of Psychology* 140, pp. 1–55.

[15] Peter Linz (2011): *An Introduction to Formal Languages and Automata*, 5th edition. Jones and Bartlett Publishers, Inc., USA.

[16] John C. Martin (2003): *Introduction to Languages and the Theory of Computation*, 3 edition. McGraw-Hill, Inc., New York, NY, USA.

[17] Mostafa Kamel Osman Mohammed (2020): *Teaching Formal Languages through Visualizations, Simulators, Auto-graded Exercises, and Programmed Instruction*. In Jian Zhang, Mark Sherriff, Sarah Heckman, Pamela A. Cutter & Alvaro E. Monge, editors: *Proceedings of the 51st ACM Technical Symposium on Computer Science Education, SIGCSE 2020, Portland, OR, USA, March 11-14, 2020*, ACM, p. 1429, doi:10.1145/3328778.3372711.

[18] Marco T. Morazán (2022): *Animated Problem Solving - An Introduction to Program Design Using Video Game Development*. Texts in Computer Science, Springer, doi:10.1007/978-3-030-85091-3.

[19] Marco T. Morazán (2022): *Animated Program Design - Intermediate Program Design Using Video Game Development*. Texts in Computer Science, Springer, doi:10.1007/978-3-031-04317-8.

[20] Marco T. Morazán (2024): *Programming-Based Formal Languages and Automata Theory - Design, Implement, Validate, and Prove*. Texts in Computer Science, Springer, doi:10.1007/978-3-031-43973-5.

[21] Marco T. Morazán & Rosario Antunez (2014): *Functional Automata - Formal Languages for Computer Science Students*. In James Caldwell, Philip K. F. Hölzenspies & Peter Achten, editors: *Proceedings 3rd International Workshop on Trends in Functional Programming in Education*, *EPTCS* 170, pp. 19–32, doi:10.4204/EPTCS.170.2.

[22] Marco T. Morazán, Joshua M. Schappel & Sachin Mahashabde (2020): *Visual Designing and Debugging of Deterministic Finite-State Machines in FSM*. *Electronic Proceedings in Theoretical Computer Science* 321, pp. 55–77, doi:10.4204/eptcs.321.4.

[23] Dominique Perrin (1990): *Chapter 1 - Finite Automata*. In Jan Van Leeuwen, editor: *Formal Models and Semantics*, Handbook of Theoretical Computer Science, Elsevier, Amsterdam, pp. 1–57, doi:10.1016/B978-0-444-88074-1.50006-8.

[24] Elaine Rich (2019): *Automata, Computability and Complexity: Theory and Applications*. Pearson Prentice Hall.

[25] Arnold Robbins (2015): *Effective Awk Programming (4th Ed.)*. O'Reilly Media, Inc., USA.

[26] Susan H. Rodger (2006): *JFLAP: An Interactive Formal Languages and Automata Package*. Jones and Bartlett Publishers, Inc., USA.

[27] Susan H. Rodger, Bart Bressler, Thomas Finley & Stephen Reading (2006): *Turning automata theory into a hands-on course*. In Doug Baldwin, Paul T. Tymann, Susan M. Haller & Ingrid Russell, editors: *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2006, Houston, Texas, USA, March 3-5, 2006*, ACM, pp. 379–383, doi:10.1145/1121341.1121459.

[28] Michael Sipser (2013): *Introduction to the Theory of Computation*, 3rd edition. Cengage Learning.

[29] John Sweller, Jeroen J. G. van Merrienboer & Fred Paas (1998): *Cognitive Architecture and Instructional Design*. *Educational Psychology Review* 10, pp. 251–296, doi:10.1023/A:1022193728205.

Programming Language Case Studies Can Be Deep

Rose Bohrer
Worcester Polytechnic Institute
Worcester, MA, USA
rbohrer@wpi.edu

In the pedagogy of programming languages, one well-known course structure is to tour multiple languages as a means of touring paradigms. This tour-of-paradigms approach has long received criticism as lacking depth, distracting students from foundational issues in language theory and implementation. This paper argues for disentangling the idea of a tour-of-languages from the tour-of-paradigms. We make this argument by presenting, in depth, a series of case studies included in the Human-Centered Programming Languages curriculum. In this curriculum, case studies become deep, serving to tour the different intellectual foundations through which a scholar can approach programming languages, which one could call the tour-of-humans. In particular, the design aspect of programming languages has much to learn from the social sciences and humanities, yet these intellectual foundations would yield far fewer deep contributions if we did not permit them to employ case studies.

1 Introduction

The introduction is structured into four subsections, respectively addressing motivation (Section 1.1), context (Section 1.2), pedagogical philosophy (Section 1.3), and contributions (Section 1.4).

1.1 Motivation

As programming languages (PL) educators, it is important to reflect on fundamental design decisions about course contents, the methodologies taught to students, and which intellectual traditions our courses will build on. These decisions are important because, over time, they shape what the field of programming languages *is* and *who* chooses to participate in the field, even perhaps shaping who does or does not feel they belong in PL classrooms. In the author's classroom experience, many PL students are interested in the study of the *design* side of PLs, yet find themselves with limited resources to study it. The thesis of this paper is that when PL case studies are used to teach *design* specifically, they provide irreplaceable contributions to PL pedagogy, compared to other uses of case studies in PL courses. To support this thesis, the heart of the paper consists of a presentation of several key case studies from the author's *Human-Centered Programming Languages* (HCPL) curriculum.

1.2 Context

This subsection provides context on the historical debate surrounding the role of PL case studies in PL pedagogy, to demonstrate why their use may be contentious to some readers.

Once upon a time, the gold standard of programming languages education was to take a tour of different programming paradigms by taking a tour of languages from each paradigm. Imperative programming might be represented by C, object-oriented programming by Java, and logic programming by Prolog. Functional programming languages could be represented by anything from a Lisp, to a member

Stephen Chang (Ed.): Trends in Functional
Programming in Education (TFPIE)
EPTCS 405, 2024, pp. 56–79, doi:10.4204/EPTCS.405.4

© Rose Bohrer
This work is licensed under the
Creative Commons Attribution License.

of the ML family (OCaml, Standard ML), to Haskell. Yet there are vast differences between these languages: most Lisps are dynamically typed and strongly emphasize macros, the ML family is statically typed and impure with strict evaluation, and Haskell is statically typed and pure with non-strict evaluation. The simple problem of identifying a PL to represent each paradigm reveals a major weakness in this course structure: paradigm labels do not neatly and usefully classify modern programming languages. For example, the language Scala is undoubtedly functional and shares much with the ML tradition, featuring heavy use of type inference and type polymorphism, with strict evaluation. At the same time Scala is undoubtedly an object-oriented language, with its Java interoperability a defining feature. Conversely, Java is statically-typed with eager evaluation and now possesses first-class functions. Yet, few functional programmers would declare Java to be a functional language, in part because the communities of Java programmers vs. Scala programmers have radically different cultural traditions. It has become clear with time that the notion of classifying languages by paradigms falls apart under any close analysis, one of many reasons this approach has waned in popularity over the years.

One of the clearest criticisms of the tour-of-languages approach is Shriram Krishnamurthi's essay *Teaching Programming Languages in a Post-Linnaean Age* [34], which advocates instead for a *tour-of-features* approach, embodied by his textbook *Programming Languages: Application and Interpretation* (PLAI) [33]. In this approach, a lesson might address a feature such as polymorphism, Hindley-Milner type inference, or lazy evaluation, but never a specific real-world language that exhibits these features. Courses using this approach often assign the implementation of each feature as classwork. That implementation often takes place in a Racket-based environment, writing interpreters for a pedagogical language with a Lisp-style syntax. The textbook contains ample example programs using each feature, and writing such programs can be included as part of coursework.

To his credit, Krishnamurthi acknowledges the tradeoffs inherent in each approach, which reflect a tension between inductive and deductive learning styles. The use of concrete existing programming languages can help students learn inductively from pre-existing example languages, while a tour-of-features places greater emphasis on deductive learning from the start, then adds example programs in the implemented language in an effort to meet both learning styles.

1.3 Pedagogical Philosophy

The subsection characterizes the pedagogical philosophy of the HCPL project, of which this paper is one part. We contrast different course philosophies as different kinds of "tours": the *tour of features* (PLAI), the *tour of paradigms* (pre-PLAI), and the *tour of humans* (HCPL). Not all texts are organized as tours, and many [24, 47, 48, 50] emphasize theory to the exclusion of other topics. Though such texts remain cornerstones of the field, it is essential not to define competing intellectual foundations as "outside PL"; given PL's status as a prestige subfield [35], such rhetoric serves to illegitimize such competing approaches.

The HCPL curriculum simultaneously owes a great debt to PLAI's *tour-of-features* approach yet aims to overcome a substantial limitation of pre-existing approaches in general. In particular, the designers of most contemporary curricula [21, 24, 33, 39, 46, 47, 48, 50], PLAI included, recognize both the importance of PL *design* and the importance of "teaching to the 90%," working to define a broad potential audience for a PL course. I wish to argue that these noble goals, which HCPL shares, fundamentally contradict the move away from case studies represented by the *tour-of-features*. Case studies are at the heart of any intellectually serious analysis of *design*, whether that analysis finds its foundations in the social sciences or in the humanities. In our efforts to ensure an intellectually serious treatment of the mathematical foundations of PLs, we must not forget that design deserves this same seriousness.

To treat both the mathematics and design of languages with proper respect in a single course is no small feat, yet case studies are the key ingredient that makes this possible, serving as the basis of the *tour-of-humans* philosophy. The HCPL curriculum is organized as a tour of different intellectual traditions for PLs, i.e., a *tour of the different humans who study PLs*. This philosophy is inseparable from HCPL's use of case studies: the lesson surrounding each case study explores how specific intellectual traditions would engage with the design of each language. The fundamental challenge of the HCPL project is that it must cover much breadth and sacrifice depth; the tour-of-humans helps maintain a coherent course structure while moving from case study to study, all while putting scholars of proofs and of humans on equal footing. Without case studies, covering this breadth in a single course would be daunting.

Moreover, the human-centered side of PLs cannot be studied without attention to actually-existing languages as they are actually used, and thus cannot be studied properly in the absence of case studies. Because HCPL also incorporates foundations and implementation, however, these case studies do not serve to eliminate rigor, instead reinforcing the foundations of programming languages while also integrating with foundational techniques in the social sciences and humanities. For detailed information on the connections drawn and the intellectual depths of each case study, proceed to Section 4 and consult Table 2 on page 64.

1.4 Contributions

The core contribution of this paper is a detailed description of five case studies from the Human-Centered Programming Languages (HCPL) curriculum, detailed in Section 4. Though the same underlying case studies appear in the HCPL textbook [5], the presentation is thoroughly different. The textbook addresses each case study in a pedagogical style designed for student consumption, whereas the present paper deconstructs each case study for an audience of PL educators, pulling back the curtain to reflect on how each one fits into the curriculum.

Secondarily, this paper contributes argumentation and reflection, assembling the case studies to make a broader argument about the potential role of case studies in PL pedagogy and reflecting on the fluid, evolving definition of the field of PL as a whole.

Positionality Statement I discuss my positionality to provide context to the analyses present in this work. The HCPL project addresses issues of gender and disability as they interact with PLs. The treatment of these topics is informed by experience as a binary transgender woman with multiple disabilities, both physical and developmental: Ehlers-Danlos Syndrome and neurodivergence. Identity is not a claim to authority, but lived experience is a valuable resource in interpreting academic scholarship and synthesizing it into a curriculum. Academic background also provides useful context: the author's PhD is in theorem-proving with substantial elements of PL theory, yet she has increasingly added both human-computer interaction and CS education research to her research agenda. Thus, the HCPL curriculum is developed from the perspective of a Computer Scientist working outward to connect with other disciplines.

Moreover, the goal of inclusive pedagogy will never be achieved by centering any single person with a marginalized identity, but only through a diversity of perspectives that can draw on our community's wide range of intersecting identities. Readers should be aware of other marginalized identities whose life experiences are essential to a complete HCPL curriculum, yet the author cannot speak to the experiences of these identities. Notable examples are native language, race, and socio-economic status. The author is a native-English-speaking, white American employed as a tenure-track professor, with the socio-economic privilege that provides. Of these factors, it is established in the literature that native

language is an important factor in inclusive PL design [58]. Though I am unware of specific publications on the latter topics, racism and socio-economic exclusion are common topics of informal conversation with my colleagues in the context of the PL community and higher education more broadly. I call for the increased publication of studies which can draw on these life perspectives and other intersectional identities that the author lacks.

I outline the remaining sections. Related work is in Section 2. Background information on the design of the HCPL curriculum is given in Section 3. The heart of the paper is Section 4, which presents the case studies. The conclusion is Section 5.

2 Related Work

Human-Centered Programming Languages This work is part of the broader *Human-Centered Programming Languages* project [5, 4]. In addition to the book itself [5], an initial companion paper describes the principles behind the book's construction, the origins of the archetype-based approach, book contents, and insights from data analysis of course reports [4]. Though that paper lists the case studies from the book, they are not documented in depth. The integration of Rust in the HCPL course is informed by the same author's prior Rust curriculum proposals [3].

In contrast, this paper focuses exclusively on the HCPL case studies. It is our goal to supplement the textbook itself by giving educators a clear picture of how the case studies can be used in the classroom, what kinds of activities might be used in such lessons, and how they fit together with the rest of the course. In doing so, this paper also serves to argue that these case studies can be used to reinforce the fundamentals of design especially, and even the fundamentals of PL theory. By design, this paper recounts the specific case studies of HCPL at great length, with the aim of arguing this broader claim.

User-Centered PLs and Programmer Experience The HCPL project is far from the first study of the relationship between humans and programming languages. A substantial body of work exists under two labels: *User-Centered PLs* [42] and *Programmer eXperience* (PX) [40], with the latter label emphasizing the inclusion of tooling beyond the language definition itself. Notable works within this area include the Whyline [30], a debugger with explanations, and the study of natural programming [43], i.e., based on programming user studies where the subjects have no prior conceptions of programming to bias their experiences. In recent years, this line of work has been operationalized in the PLIERS design framework [12].

The distinguishing feature between HCPL and (the majority of) User-Centered PL and PX approaches is that the latter address the human element of programming primarily through the notion that a programmer is a user of a software system for the task of programming. As a result, these approaches are rooted in HCI, more specifically in user experience (UX). In contrast, the focus of HCPL is on the breadth of different humans who study PLs, the archetypes, which also distinguishes us from the wide array of existing high-quality PL textbooks [21, 24, 25, 33, 39, 46, 47, 48, 50]. This intentionally includes not only perspectives on usability, but critical perspectives from the humanities, while also putting these views into conversations with approaches to PLs rooted solidly in theoretical computer science. That said, the perspectives represented in the User-Centered PL and PX communities are not homogenous, and some authors' perspectives align more closely with that of HCPL. For example, Ko [29] invites viewing PLs as socio-technical systems; as is HCPL, this is fundamentally an invitation to a highly interdisciplinary perspective.

Pedagogy The design of HCPL is informed by a number of concepts in pedagogy scholarship. A main strategy for helping students navigate the technical breadth of the course is by providing flexibility and transparency in the evaluation process. An early iteration of the course took a fully project-based [32] approach, before pivoting to a blend of autograding and completion[1] grading with peer-review elements in subsequent instances of the course.

These pedagogical design decisions are informed by the philosophy of ungrading [57, 31, 17], which views rating and ranking of students as fundamentally detrimental to the process of learning, aiming instead to maximize intrinsic motivation, often through choice and ownership. In ungrading contexts, a common challenge is student self-regulation [63], a trait which is associated with future student success but can rarely be cultivated by a single course. In response to this challenge, the present iteration of the HCPL course uses autograding as a means of providing external regulation of students, under the hypothesis that once students have fulfilled one set of classwork obligations thanks to external regulation, they are more likely to fulfill their completion-graded assignments as well. See prior work [4] for further discussion of the relationship between HCPL and ungrading, including student reactions to the practice, both positive and negative.

HCPL is not the only open-access PL textbook nor the only to provide open-access companion materials, but the choice of open access was an intentional one. Open education [23, 27, 53, 61] is the long-running movement to make educational as widely available as possible through the reduction of barriers. The use of open access and even, in part, the use of autograding are intended to enable an open education experience, e.g., for self-study outside a university setting.

The HCPL course is potentially challenging to execute for instructors because of its technical breadth. One strategy for coping with its breadth is to keep an explicit focus methodology, and bring students back to the different methodologies used. In written tasks from completion-graded assignments, the literatures on reflective journals [22] and autoethnography [15] can both guide classroom practice. In the social science modules of HCPL, key techniques include the use of thematic analysis [6, 10] and personas [45]; ample resources are available for both.

Other teaching methods in HCPL have their origins in the humanities and have seeped into CS over time. In particular, the use of dialogues between archetypes has thousands of years of methodological precedent in the form of Socratic dialectic [14, 2], though our dialogues by comparison are intended to reveal a multitude of co-existing perspectives instead of guiding a student to a specific truth known by a teacher. In the modern era, *Proofs and Refutations* [36] is a well-known use of dialectic to teach computer scientists about mathematical proof, which is at the heart of PL theory. Educators firmly within the humanities can dive into more specific methodologies not addressed in depth by the HCPL text, such as codework [38], poetry in the medium of code.

The HCPL approach is also indebted to the literature of PL pedagogy specifically, not limited solely to Krishnamurthy's essay on the role of paradigms in PL education [34]. For example, recent scholarship on social issues education for CS students has advocated integration across the curriculum instead of standalone courses [13]; HCPL's coverage of gender and disability issues implements this approach. The use and discussion of toy teaching languages in HCPL is informed by the lessons learned by other PL education researchers; classroom discussions of the transition from a teaching language to a production quality language are partially inspired in particular by *A Data-Centric Introduction to Computing*'s [18] explicit coverage of this transition process.

Like any new course design, the development of HCPL was motivated by the dearth of existing

[1]Completion grading is a grading scheme where full points are awarded on all problems that are completed, irrespective of any quality assessment

courses that filled its interdisciplinary niche. At the same time, the few prior courses that filled this niche are thus all the more important. In particular, the work of Michael Coblenz, Jonathan Aldrich, and their collaborators [11] directly influenced the first iteration of the HCPL course, an influence which thus carries through indirectly in the book.

Case Studies Most of the case studies discussed herein are already documented in the research literature. I cite this prior documentation, which is the basis of my own work, respectively for Penrose [62], Torino [41], FLOW-MATIC [26, 59], Processing [49], and Twine [20].

The C-Plus-Equality case study is the exception; I am not aware of any prior peer-reviewed literature engaging with this case study, and minimal scholarly writing about it any form [60]. By helping bring this case study into the literature, I hope to raise awareness among PL researchers of gendered harassment that has used the vocabulary of PL. To highlight the importance of documenting this case study: the full case study materials were publicly visible on GitHub [19] when I began preparation of this manuscript, but the repository was disabled by GitHub[2] by the time of submission. Beyond this specific case study, the discussion of gender in HCPL has been informed by various scholars [7, 8, 9, 52].

3 Course and Book Structure

This paper is dedicated to the case studies of the HCPL curriculum, arguing for their pedagogical validity and documenting them in the process. For detailed documentation on the HCPL course, see prior work [4]. Here, we provide relevant background information about the course's contents in Section 3.1, its use of *archetypes* in Section 3.2, and the coursework in Section 3.3.

3.1 Contents

I list the chapter titles, length estimates[3], and languages discussed in Table 1 on page 62. Many chapters contain topics which could be viewed as case studies; I list which case studies were included in this paper and briefly discuss how they were selected.

Given the interdisciplinary nature of the course and textbook, I am often asked how the book's contents align with more traditional intradisciplinary courses. My courses are housed in a CS department at an institution which does not segregate HCI from CS, thus it contains substantial elements of both CS (Ch. 1–9) and HCI (e.g., in Ch. 10–14 and 18). For instructors concerned about integrating an interdisciplinary course into a disciplinary setting, one potential strategy is employing a subset of the HCPL text in concert with other materials.

As Table 1 on page 62 shows, not all of the HCPL case studies are covered in this paper. I used the following considerations in choosing which case studies are presented in the current paper. I excluded case studies where I believed there is substantial debate (e.g. between the archetypes) about whether it *is a case study*, such as PEGs and process calculus. I excluded case studies where writing about them would substantially duplicate another case study, such as Inform and Penrose. Lastly, I prioritized case studies with little existing documentation in the academic literature by including C+= which is ill-documented and excluding Randomo, which is well-documented.

[2] The repository was disabled for terms-of-service violations due to its misogynistic content. My analysis aims to shed light on that misogyny.

[3] The length estimates are provided automatically by the publishing platform, Bookish.press. Reading time varies by reader, but these estimates give a sense of relative length.

Table 1: Table of contents for current iteration of book

No.	Title	Length	PLs	In Paper?
1	Introduction	10min		
2	What is a Language	25min		
3	Programming in Rust	1hr	Rust	N
4	Regular Expressions	35min	RE	N
5	Context-Free Grammars	50min	CFG	N
6	Parsing Expression Grammars	30min	PEG	N
7	ASTs and Interpreters	35min		
8	Operational Semantics	5min		
9	Types	50min		
10	Users and Designers	40min		
11	Quantitative Methods & Surveys	45min	Randomo	N
12	Qualitative Studies	50min		
13	Gender	35min	C+=	Y
14	Disability	30min	Torino	Y
15	Media Programming	15min	Processing	Y
16	Play	15min	Twine	Y
17	Natural Language	25min	FLOW-MATIC,Inform	Y,N
18	Diagramming	15min	Penrose	N
19	Process Calculus	20min	Proc. Calc.	N
20	Cost Semantics	20min	Par. ML	N

3.2 Archetypes

In arguing our thesis that PL case studies can be deep, an awareness of course structure is essential. The HCPL course structure is called the *tour-of-humans* structure, where students tour between different *archetypes* representing different scholarly perspectives. The five archetypes are named the Practitioner, Implementer, Theorist, Social Scientist, and Humanist. I give a subjective listing of the chapters in which each archetype takes a substantial role: Practitioner (1–3,15–18[4]), Implementer (4–7), Theorist (8,9,16,19,20), Social Scientist (10–14,18), Humanist (13–15,17). In the text, the archetypes are used to explain motivations and approaches to solving problems. They are put into conversation with one another using dialogues within certain chapters.

To contextualize the overall HCPL course design, I summarize each archetype. I do so by characterizing who they are, what questions they ask, how students interact with them, and how the archetypes relate to the case studies.

The Practitioner: The Practitioner is someone who interacts with code as a programmer, but does not implement, design, nor theorize about programming languages. The Practitioner's fundamental question is "How do I write this program?" Students engage with this archetype by writing code and reflecting on their experience, including classroom activities with case study languages.

The Implementer: An Implementer is anyone who implements a programming language, typically as a compiler or interpreter. The Implementer's fundamental question is "How do I implement this programming language?" Students engage with this archetype by implementing PLs in coursework.

The Theorist: The Theorist is anyone who does the work of defining PLs as formal languages or

[4]Lessons 15–18 involve classroom coding exercises in case study languages

analyzing them mathematically. To them, 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 powerful theorems about the correctness of programs. The Theorist's fundamental question is "What can I prove about this language?" Based on my students' needs, I limit engagement with the Theorist to reading and contextualizing proofs and mathematical notations. Case studies (e.g., Twine) engage with the Theorist by highlighting practical applications of theory.

The Social Scientist: The Social Scientist is someone who undertakes rigorous academic study of humans. The Social Scientist's fundamental question about programming languages is "How do programming languages affect communities of people?" Students engage with the Social Scientist by designing self-directed user studies and performing them on classmates. The case studies engage with the Social Scientist both by teaching social science methodologies (Randomo) and by showcasing their potential contributions to novel designs (Torino, Penrose).

The Humanist: The Humanist also studies humans, thus I am often asked how and why HCPL distinguishes the Humanist from the Social Scientist. Both archetypes are employed in HCPL because PL design can benefit from motivations and methodologies firmly within the humanities, particularly the use of critiques driven by careful textual analysis. As with all the archetypes, real-world Humanists are more varied than a single character can portray. For our HCPL archetype, however, I restrict the Humanist's fundamental question to be "How can social analysis be applied to PLs?" Students engage with the Humanist through a self-reflection exercise drawing on their own experience and by following lectures which employ humanistic techniques. Case studies engage with the Humanist through the use of historical methods (FLOW-MATIC) and hermeneutic inquiry (C+=), and by drawing on the model of the disability spectrum from disability studies (Torino).

The Social Scientist and Humanist receive particular focus in this paper; for deeper discussion of the roles of the other archetypes, see prior work [4].

3.3 Coursework

Successful pedagogy with the HCPL approach hinges on flexible design of coursework. As of this writing, the course uses a "*two-spoke*" model, where the first spoke consists of auto-graded programming assignments and the second consists of completion-graded written assignments with student peer-review. As of the Fall 2023 iteration of the course, the programming assignments prioritize the Implementer and the written assignments prioritize primarily the Social Scientist, followed by the Humanist. The Theorist is assessed indirectly in programming assignments which rely on understanding of parsing, type systems, and evaluations, but receives the most focus in exam questions that directly address theory.

The current course iteration has five assignments, each divided into a programming and written assignment with the same deadline. The programming assignments are: a Rust language warmup, a parser, an interpreter, a type-checker, and a small library of quantitative data analyses. The respective written assignments are: a questionnaire on students' personal learning goals, an initial brainstorm for student-directed PL user studies, a reflection on personal experiences of Rust, a full user study proposal, and a writeup of user study results.

What is the relationship between the coursework and the case studies? On the one hand, these assignments do not require knowing case study details—the Fall 2023 course assesses factual case study knowledge through a final exam instead. However, the course's core social science skills are refined throughout the lectures by their repeated appearances throughout the case studies, and the written assignments fundamentally require these skills. Though case study details are not tested on homeworks, the case studies still have the critical role of preparing students for their homework. It is natural for stu-

dents to wonder how students engaged with written homeworks given that they are graded by completion and that the design material may be very new to many students. In the Fall 2023 undergraduate course, students showed substantial creativity, such as the development of text adventure games to employ in their user studies, bringing musical instruments to class to explore musical PL interfaces, and physical activities to test the role of posture in programming. These classroom experiences are consistent with the common assertion by *ungrading* [57] proponents that creativity increases when grading is removed but intrinsic motivation remains.

4 The Case Studies

We now arrive at the heart of the paper. In this section, I present the HCPL case studies. For each case study in turn, I argue the thesis that these case studies reinforce foundational concepts in support of key learning goals, each with specific design implications Colloquially, I argue that PL case studies can be deep.

In Table 2 on page 64, I list out each case study, the topic(s) it connects with, and its *depth*, i.e., which concept or methodology is fundamental to the case study analysis, each with a design implication.

Case	FLOW-MATIC	Processing	Twine	Torino	C+=
Connection	History	Media Arts	IF	Disability Stud.	Gender Stud.
Depth	Sources	Them. Anal.	FSMs	Disab. Spectrum	Hermeneutics
Implication	Audience	Continuity	Winnability	Hybrid Syntax	Commun. Discourse

Table 2: Connections, Depths, and Implications of Case Studies

The FLOW-MATIC case study connects with the field of history, its depth lies in collection and interrogation of primary sources. Its design implication is that FLOW-MATIC's designers did identify a specific audience, enabling us to apply contemporary audience-driven analyses, despite the fact that modern design methodologies had not yet been formalized at that time. The Processing case study connects on the surface with the field of media arts, but its methodological depth is the social science method of thematic analysis; throughout this lesson I show how a complex mix of written text, visual artifacts, and lived experiences can be used to form our opinions as designers, going beyond the more basic or rigid forms of thematic analysis. A key design implication is that the notion of continuity between languages is two-sided, and designers ought to be aware of which languages might be learned *after* a programmer has learned their language. The Twine case study connects on its surface with the topic of interactive fiction (IF), yet its methodological foundations are solidly within CS theory: finite state machines (FSMs). The direct design implication is that Twine's structure enables static analysis of Twine games' winnability, yet the broader learning implication is that CS theory might be applied in surprising places. The Torino case study connects with the field of disability studies and its depth lies in its conception of disability as a spectrum. The design implication is that hybrid syntaxes (e.g., tactile-and-visual or aural-and-textual) support collaboration between programmers across multiple positions on the disability spectrum. The C+= case study connects with the field of gender studies and its depth lies in its hermeneutic inquiry [51, 54] approach, which embraces the notion of textual interpretation as an active process of creation. The design implication of the case study is that a designer cannot fully understand the user experience of their language without careful attention to ongoing discourse between members of the language community, including elements of discourse that must be carefully read out of the implications of a document.

We now proceed to the case studies: FLOW-MATIC (Section 4.1), Processing (Section 4.2), Twine (Section 4.3), Torino (Section 4.4), and C+= (Section 4.5).

4.1 FLOW-MATIC

This case study is an invitation to students to the study of the history of PLs. To the Humanist, the work of history is often the work of reconstructing an image that will never be complete, building narratives and arguments about the past based on the small window provided by the historical record. In the history of PLs, we often forget to teach this element, perhaps because our own history is so brief, or perhaps because we are not professional historians.

The case study on FLOW-MATIC [59] (predecessor of COBOL) seeks to expose PL students to the work of history, of reconstruction. This case study is introduced shortly after students are exposed to fundamental contemporary design tools, such as the use of user personas, design of user studies, and interpretation of qualitative data from user studies. In approaching the history of PLs, we are confronted with the fact that most well-known PLs, both historically and today, did not follow any such formal design process. This does not mean that their creators lacked all design knowledge, but rather that the process was far less formalized and recorded far less thoroughly than it might be in a modern approach. The goal of this case study, then, is making sense of surviving documents from early PL designs and extracting an understanding of their design goals and approaches that are comprehensible to a modern audience. FLOW-MATIC was chosen in particular because it is the earliest known PL which sought to mimic natural language and because connections with natural language remain a topic of curiosity for students to this day. Thus, our research questions as historians of FLOW-MATIC are: "What goals and context motivated the earliest natural-language PL design efforts? What is the legacy of those efforts, and how can this legacy inform the role of natural language in contemporary design?"

The first primary sources for this historical analysis are the recorded statements of Grace Hopper, who was the designer of FLOW-MATIC and a core figure in the COBOL design process. The following quote [26] from Hopper is used to contextualize the design goals and motivation 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; I do not know what they mean, I have not time to learn symbols'

This quote serves as an opportunity to show a direct connection between foundations and case studies. A previous foundational lecture uses the ISO 9241-11 [56] definition of usability to teach students what it means to define a usability problem and assess a proposed solution. Students are taught to consider three questions of problem definition and three of assessment, respectively: defining users, goals, and context, and assessing effectiveness, efficiency, and satisfaction.

In teasing apart Hopper's statement, I show that it addresses elements of usability that are echoed by a standard released decades later. She provides a clear definition of the user population: businessmen (and, in the full statement, military staff), which carries with it implications about goals and especially context. Her quote reflects the idea that satisfaction was the root concern and that effectiveness had been hindered by dissatisfaction; perhaps efficiency was not the top design priority. The unwillingness to use symbols is also related to discussions of programmer self-efficacy, which arises also in the lecture on gender.

Our second primary source gives students a chance to engage hands-on with specific code. Official, contemporaneous documentation of FLOW-MATIC [59] has been well-preserved by historians of computing, including source code of example programs. The following program text is presented in the classroom and students are given time to read it and guess at its meaning before further instruction:

```
INPUT INVENTORY FILE-A PRICE FILE-B ; OUTPUT PRICED-INV FILE-C UNPRICED-INV
  FILE-D ; HSP D .
1  COMPARE PRODUCT-NO (A) WITH PRODUCT-NO (B) ;
   IF GREATER GO TO OPERATION 10 ;
   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)
```

It is common for students to struggle to discern the meaning of this program (updating prices of items in an product inventory) despite their substantial programming experience. In experiencing this struggle, I highlight that the use of natural language syntax is no guarantee that programmers will find a language usable, and that usability can only be understood relative to a given societal context. We use this opportunity to explain why the program makes extensive use of `GO TO` and `JUMP TO` operations — structured programming had not yet been invented, yet is such a standard feature today that the use of `GO TO` is by far the exception rather than the rule. The familiarity of natural language does not override the unfamiliarity of pre-structured programming, nor did FLOW-MATIC's contribution of one usability feature (natural language syntax) override the many usability challenges that remained in languages of that era.

The reflection on this sample program then turns to teasing apart the different kinds of complexity that can arise in a PL syntax, distinguishing, e.g., a verbose syntax from a syntax which requires a large number of keywords or production rules. We observe historical trends in how PL designs have engaged with both forms of complexity. Over the years, verbose code has remained widespread, yet improvements in editor automation have substantially reduced the number of user actions required to edit verbose programs, automation not available in the FLOW-MATIC era. In contrast, we lack clear examples of automation for overcoming complexity of formal grammars or high keyword counts, and have instead seen languages with lower keyword counts take over in mainstream use. This observation is intrinsically linked with guiding principle of natural language, as PLs which seek to remain fully faithful

to a natural language syntax tend to have higher keyword counts and production rule counts than others. Instead, we have been left largely with languages which pull their keyword vocabularies from natural language, but make no attempt at grammaticality.

4.2 Processing

Processing [49] is a language intended for use by media artists, such as visual, audio, and video art forms, and which has also seen use in pedagogy applications. The Processing lesson comes after Natural Language. It reinforces several key themes from the prior lecture: that the work of defining an audience has concrete impacts on the designs of actually-existing PLs, that this design is culturally and historically contingent, and that textual analysis of the written historical record is a means for unpacking these contingencies. The differences between Processing and FLOW-MATIC are equally essential: most notably, Processing remains in widespread introductory-level use as of this writing, and is far more likely to be received as "familiar" by contemporary readers. It is this contrast between unfamiliar and familiar which helps students internalize the idea of language designs being located in specific times in history. In contrast to FLOW-MATIC, free web-based implementations of Processing are widely available, allowing students to experiment with Processing code in class, bringing a moment of personal experience to their analysis of the language.

The course's use of *multiple* case studies with recurring themes is itself important, because this creates space to engage with underlying ideas of methodology. The foundational lectures in design provide a primer on interpreting survey responses through the widespread qualitative method of thematic analysis [6]; the FLOW-MATIC and Processing case studies both reveal that the qualitative paradigm includes far more than just the analysis of survey data. The main "facts" presented to students in the Processing lesson are a list of themes, specifically PL design values, reconstructed from an informal thematic analysis process. Compared to the FLOW-MATIC case study, the Processing lesson in particular highlights the diversity of sources used to analyze the themes. The analysis is based on academic texts produced by Processing designers, the lived experience of the lesson author and students in writing programs, and oral communication with other Processing programmers. In contrast to FLOW-MATIC, an analysis of Processing has ready access to primary sources beyond the written record.

The themes provided for the analysis of Processing are *visuality*, *immediacy*, and *continuity*, the latter of which is explained in relation to the basic historical concept of *contingency*.

To observe *visuality* as a theme of Processing may at first seem thoroughly non-deep, contradicting the thesis of this paper. On the contrary, the depth lies in the discussions that fall out of the themes. I give one small example of how the brief hands-on experiences of programming in a classroom can add nuance to these themes. One student remarked that ce[5] found `printf`-style debugging much harder in Processing than other languages ce knew, perhaps because of its focus on visuals. This quick remark complicates the common assumption that visual languages are automatically easier to use than textual ones; this relation may flip if the user is primarily comfortable with text. We align this discussion of visuality with the core idea of PLs as interfaces. In drawing this connection, I further contrast Processing, where visuals are a core program *output*, with languages whose syntax is visual or whose inputs are images. The other aspect of depth lies in contrasting visuality from the other themes, such as immediacy.

We characterize *immediacy* as the experiential goal that when a programmer makes a change to a program (and attempts to re-run it), there is no perceptible delay between making the change and observing its impact on the behavior of the program. We align this value with the broader principle of

[5]This student's pronouns are ce/cer/cers. These are *neopronouns*, i.e., neologisms used by choice as personal pronouns, often for purposes of self-expression.

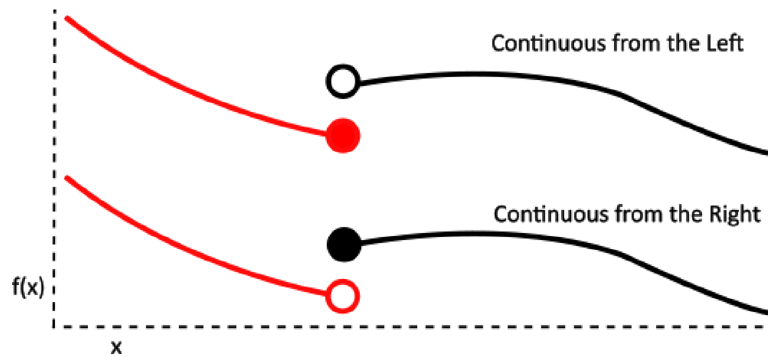


Figure 1: Diagram of left-continuous and right-continuous mathematical functions

self-efficacy and the idea that programmer motivation is tied to self-belief in the capacity to plan and execute programming tasks. Immediate feedback as provided in Processing, by shortening the iterative development cycle of a program, may (be intended to) provide an increased sense of control. We contrast several concepts with which immediacy is sometimes associated but from which it is distinct. In particular, Processing's immediate feedback is visual in nature, but immediacy is by no means restricted to visually-oriented programming environments, with interactive programming prompts (REPLs) being a longstanding PL feature. Likewise, we can separate the question of immediacy from the question of typing discipline. Many of the best-known REPLs are for dynamically-typed languages such as Python and various Lisps; because dynamic typing reduces the number of potential error messages between the writing and first execution of a program, it can be seen as a form of immediacy support. Yet this does not remove the fact that many languages with robust (e.g. Hindley-Milner-style) type systems such as Haskell and Standard ML also provide well-established REPL interfaces. The decisions of typing discipline and immediacy are neither equivalent nor fully independent.

Contingency is the basic concept that the occurrence of each historical event is dependent on the occurrence of previous events, and thus not inevitable, i.e., history could have unfolded differently. In the HCPL context, contingency is presented as the concept that if past PLs had been designed or employed differently, the design decisions of new PLs might have been made differently. The concept of contigency serves as the basis of two potential opposing design values: *continuity* vs. *radicalism*. Continuity is defined as the value that pre-existing design decisions are preferred by default until a compelling argument for a change is presented, while radicalism is the opposite value that novel designs are preferred by default. Examples of radicalism can be found in *esoteric programming languages*; Processing in contrast emphasizes continuity. Continuity is closely linked to the notion of *familiarity*: by identifying the specific user audience of a specific design, one can assess which existing design decisions are most likely familiar to the given audience, and thus decide the baseline set of design decisions against which an analysis of continuity or radicalism will be made.

The notion of continuity generalizes the notion of familiarity by employing a metaphor from a metaphor from elementary calculus: a function of a single variable could be *continuous from the left*, *continuous from the right*, both, or neither. Such mathematical functions are depicted in Figure 1 on page 68 and used as a visual metaphor in the course. Continuity from the left means that a PL is continuous with respect to the design decisions with which the audience is familiar before learning to use it. Continuity from the right means that a PL is continuous with respect to the design decisions which the audience is likely to encounter *after* learning to use it. This theme fell out of analyzing the Processing designers' writings about their pedagogical uses of the language, synthesized with the present author's

experiences using the languages Racket and Standard ML in the classroom. Processing intentionally employs many syntax choices that align closely with widespread languages such as C and JavaScript, but the design behind this decision was not so simple as an appeal to languages that students already know. In contrast, Processing is often used with *first-time* programmers, who thus have no pre-existing attachment to a particular syntax. Instead, it is is designed with the recognition that some first-time students, even those who start from media backgrounds, will pursue further computer science studies, and then *when they do so, they ought to find Processing relevant to future studies*. The Processing strategy is not to make this language popular in industrial use, but to make it so visually similar to popular languages that students can immediately recognize programming concepts learned through Processing as relevant to their future programming work.

In my own experience using languages specially designed for pedagogy (the *How to Design Programs* [16] languages), it was a common refrain among students that the languages are not useful in their lives or, when exaggerated, "not real code". Through this lens of continuity, we can reflect as educators on the breadth of strategies available to us. The author has certainly advocated to students that programming concepts learned in a teaching language are transferable, or that there is intrinsic value in learning a broad variety of PLs, yet is worth having multiple strategies at hand, and designers in the pedagogy space can use the language of continuity to explore how students might integrate their knowledge from introductory courses with later study. Indeed, in the discussion of Processing with students, I have also pointed out recent curriculum which use specialized languages for teaching yet put explicit thought into the question of transitioning into new languages afterward [18].

4.3 Twine

As PL educators, we often seek to convince students that foundational concepts will have concrete benefits in applied development work. In the functional programming context, foundational concepts could include higher-order functions, functional transformations on persistent data, metaprogramming, and type safety. Though HCPL is not solely focused on functional programs, type safety is a shared core theme, from which we reach the Theorist's core theme of seeking universal, guaranteed predictions about the behavior of a program. This theme is particularly challenging to drive home in the classroom. Often, we drive the theme home by writing specific programs in specific languages with strong guarantees, yet this has a limitation: students may conflate the details of a chosen language with the underlying principles. For example, languages with strong guarantees have sometimes scared students away with unfamiliar syntax. Yet it is hard to do better than this approach. As much as we wish we could show students the joy of inventing a new type system guarantee for the first time, this is typically a research-level task and thus out of scope.

In HCPL, we supplement the use of an advanced type system with a second approach: walking students through the discovery of theoretical foundations in a programming tool that was never intended to have them. Though the students are walked through a specific example, the application domain is far-removed enough from the foundations that the joy of discovering a new application is still captured.

Twine [20] is a graphical tool for building interactive fiction games. Although it allows inserting code in languages such as JavaScript, it does not advertise itself as a PL. Our case study on Twine shows that by looking through the lens of the Theorist, we can quickly see a new formal language and formal guarantees where none existed before.

A Twine game consists of a graph $G = (V, E)$, where vertex set V is a set of passages of text and edge set E captures transitions between passages. The Twine game engine shows a single passage $v = (n, p)$ of text on the screen at a time, where n is the title of the passage and p is its text. A player plays a game

by clicking *links* embedded in the passage, each of which brings up a new passage on the screen. Each link e can be modeled as $e = (u, v, \ell)$ where u is the name of the passage where the link appears, v is the name of the destination, and ℓ is the highlighted text on which the link appears within the passage[6]. We assume that for a given pair u, ℓ, the value v is unique. Let E stand for the set of all such edges, then a Twine game G comprises $G = (V, E)$ and the current state s of a game at any moment is captured by a name $s = n$ such that $(n, p) \in V$ for some p.

To the Theorist, these definitions scream out "I am a graph!" The labels of this graph represent a single input action out of an input sequence, suspiciously similar to the workings of a deterministic finite automaton (DFA). At this point, the lesson turns to explore a more precise question: "How does Twine relate to a DFA and what guarantees will follow?" The fundamental idea is that the winning plays (input sequences) for a Twine game will correspond exactly to the accepted strings (language) of a DFA.

Recall that a DFA is defined as a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the set of possible states, Σ the set of input actions, δ a function describing all possible state transitions, q_0 the inital state and F the set of desired final states (accepting states). The HCPL curriculum includes review of the foundations of regular and context-free languages, preparing students to explore Twine as DFAs. Only q_0 and F are not inferable from the graph structure G; we treat these values as parameters which would be supplied in addition to G. This is a small assumption because every Twine game does specify a specific start passage and many Twine game designers consider some end states to be "good" and others "bad." We define Q, Σ, δ in terms of G, recalling $\delta : Q \times \Sigma \to Q$:

$$\begin{aligned} Q &= \{n \mid \exists p.\ (n, p) \in V\} \cup \{\mathsf{err}\} \\ \Sigma &= \{\ell \mid \exists u, v.\ (u, v, \ell) \in E\} \\ \delta(q, \ell) &= \text{the unique } q' \text{ s.t. } (q, q', \ell) \in E, \text{ else } \mathsf{err} \end{aligned}$$

where err is a distinguished error state not appearing in G.

Having completed this definition, we introduce a core methodological concept: "Once you notice that language X matches foundation Y, review what facts are known about Y and attempt to apply them to X." We apply this approach using two of the most fundamental facts about DFAs: 1) regular languages are the languages accepted by DFAs and 2) the pumping lemma characterizes the expressive power of such languages. Recall the pumping lemma:

Lemma 1. *Let m be a DFA $(Q, \Sigma, \delta, q_0, F)$ and let $n = |Q|$ be the size of Q. Let $s \in \Sigma^*$ be any string whose characters are elements of Σ and let the length $|s|$ of s be at least n.*

Then s can be decomposed into three consecutive subsequences $s_1 s_2 s_3$ such that $|s_2| \geq 1, |s_1 s_2| \leq n$ and for all natural numbers k, we have that $s_1 s_2^k s_3$ is accepted by the DFA.

The pumping lemma directly allows inferring upper and lower bounds on the length of accepted strings; in Twine, this immediately yields bounds on the lengths of winning plays:

Theorem 1. *The number of moves in the shortest winning play, if any exists, is less than the number of passages in the game.*

Theorem 2. *Let n be the number of passages. If there exists a winning play with n or more moves, there exist arbitrarily long winning plays.*

On an even more fundamental level, the unpacking the definition of regular languages (as the languages of DFAs) gives an exact characterization of the expressive power of Twine games:

[6] Because the same text might appear multiple times in the passage, this is not how a production-grade implementation of Twine works, yet it is a useful simplification for presentation purposes.

Theorem 3. *For every Twine game G, there exists a regular expression which accepts exactly the winning plays of G.*

These guarantees are tangible enough to "feel real" to students, yet have a close tie with a familiar foundation (DFAs) to feel "theoretical" at the same time, supporting the goal of convincing students to invest in theoretical thinking.

The full lesson also develops an operational semantics for Twine, in order to reinforce prior lessons on the use of inference rules, yet the core takeaway holds with or without operational semantics.

4.4 Torino

This case study uses the PL Torino [41] to illustrate how concepts from disability studies can inform PL design. Designing for disability and accessibility has a rich scholarly tradition which we will not try to recount in detail. We merely note that mainstream design research has not always centered critical perspectives from disability studies nor the lived experiences of disabled people; for a selection of research works which do, we refer readers to a recent workshop from the conference CHI [55].

This lesson emphasizes the principles of disability as a spectrum, the tension between visibility and invisibility of disabled status, and to a lesser extent, self-determination of disabled people. Lesson objectives regarding the disability spectrum include assessing PL design decisions for their usability for programmers whose disability symptoms vary substantially from day to day, or which are used on teams of programmers with differing disability levels. The lesson objectives for visibility include identifying PL design decisions which force a disabled person to be visible or invisible vs. those which give the person a choice.

The case study on Torino [41], supports these learning objectives. A key way it supports the objectives is by highlighting that the notion of disability as a spectrum had fundamental impacts both on the syntax of Torino and on how its designers structured their user studies. It does not directly support the learning objectives about self-determination; those objectives are supported through brief case studies about the research works and life experiences of disabled researchers in the PL community.

Torino was a research project which developed a PL for use by visually-disabled children ages 7–11 in a classroom environment where the level of vision varies substantially between students. As with other PLs targeting this age range, it is not the goal to teach students to write full-fledged applications. Instead, typical goals for this group range from understanding of computational thinking principles to promoting interest and sense of belonging in computing, with an eye to robustifying the CS education pipeline. A major challenge in teaching this age group is their limited literacy, which makes (e.g., block-based) visual PLs a standard approach for reaching them. This is the fundamental challenge Torino addresses: if visual programming is the standard for children's PLs, what is a designer to do when vision cannot be relied on?

Torino's solution to this challenge is a *tactile*, i.e. touch-based, syntax. The block-based paradigm is transformed into a bead-based paradigm. Every basic PL construct corresponds to distinct type of physical "bead," each with its own shape that can be told apart by touch alone. In contrast to virtual wires in block-based languages, physical wires are used to connect the input and output ports of each bead. The beads include operations for playing and modifying sounds, as opposed to creating visuals. The above description illustrates the novel designs that emerge when we focus on accessible design, but does not demonstrate the lesson objectives. We address these next.

Of the lesson objectives, Torino most directly speaks to our objectives regarding the disability spectrum. It is a crucial design decision that in Torino, the different bead types have both a distinct shape and a distinct visual appearance, including different colors. A majority of legally-blind people have some

sight, thus accommodating the use of vision is not a tangential goal, but a central one. On the contrary, designers must remember broader classroom goals for this age group: social and emotional development are key goals for all students in this range, and are often a heightened concern for children whose disability status may lead to exclusion in other spheres of life. By creating a language which can use both touch and sight for interaction, the Torino designers support the goal of allowing fully blind and partially-sighted disabled children to play and work together, which is supportive of their social development. At this point, the lesson turns to show that this understanding of the disability spectrum is not only fundamental to the design of the language syntax (a tactile syntax with visual elements), but also essential in designing user studies. The user studies performed in the Torino project specifically explored how fully blind and partially-sighted children work together, e.g., collaborating on a single program. These studies showed that both groups used the language effectively, but in different ways, with fully blind children physically scanning the beads to identify points of interest and partially-sighted children relying on sight to locate specific beads. This detail demonstrates that principles from disability studies help shape which questions a researcher or designer even asks, let alone the design solutions they arrive at.

The lesson then explores the tension between visibility and disability, i.e., the tension of whether to disclose one's disabled status to people in daily life. Though Torino's designers do not discuss this issue explicitly, the discussion of the disability spectrum in Torino provides a strong foundation for this next objective. In the same classroom, some students' disabilities may be more visible than others, and the classroom context is itself a key piece of the puzzle. Compared to society at large, the classroom is likely a relative safe space where the adults in the room are educated and supportive regarding disability issues, and it may be a space where children are willing to let down their masks and be visible in their disability, an act which provides space for community-building and support, yet represents vulnerability. The lesson segues into how the same underlying design principle, allowing multiple modes of interacting with a single program, actually supports the agency of disabled people in choosing whether to be visible, i.e., whether or not they choose to mimic the programming approaches used by abled programmers. To support the final goal, the lesson closes with highlighting the work of a visually-disabled researcher on languages which support multiple (i.e., hybrid) syntaxes [1].

4.5 C-Plus-Equality

Content Notice: This section discusses misogyny and transphobia.

The Gender lesson opens with a case study named C-Plus-Equality (C+=) [19]. In contrast to the other case studies of HCPL, this case study is brief and is purely motivational in nature. To my knowledge, it is also the only case study which has never had its semantics formally defined nor had a production-quality implementation. This makes it no less important and no less deep. Before exploring the depth of this study, we must frame its history.

The language C+= is not a "PL" in the traditional sense. Rather, it is a programming language design proposal, distributed in the form of a GitHub repository, containing a minimal prototype implementation using the C preprocessor, design documents, and example programs. The repository lists its author as "The Feminist Software Foundation," but has been attributed by online observers [37] to one or more misogynist trolls organized through the anonymous image board 4chan[7]. As of this writing, C+= is not new, with the repository dating to 2013.

Its visibility and influence have waxed and waned over time, however. As of the writing of HCPL, simple web searches about the intersection of gender and PL design still return C+= as the top result,

[7] 4chan, like the other chan boards, is a forum for conversation on a wide variety of topics, but is arguably best-known for its role as an a organizing space for far-right social activists.

far before the relevant works of HCI scholars like Burnett [8], artists like micha cárdenas [9] and Mez Breeze [7], or even prominent scholarly discussions within the humanities, e.g., via the HASTAC [52] blog. The latter example is especially noteworthy because online observers [37] have cited C+= as a *reaction to* the HASTAC blog post [52]. This anecdote about search results is used as the opening motivation to the Gender lesson, as a way of uniting students who might otherwise be divided in their beliefs about society. Not all of the author's students are feminists, but few would dispute that their feminist classmates deserve access to intellectually-sound curriculum, and thus even more-conservative students tend not to object to the lesson's presence.

The depth of this case study lies in the fact that our classroom analysis of the language relies on foundational perspectives techniques from the humanities, and cannot be properly understood using the perspective of any other archetype. Specifically, we unpack this case study by starting with the prompt: "Can a PL Spread Misogyny?" and remarking on the nature of the question. Few computer scientists have ever been asked a question of this nature, yet similar questions are routinely asked in *media studies*, i.e., they are frequently asked about popular media such as songs, movies, TV shows, or novels. A key insight of this case study is that PL design documents, implementations, and code more broadly can all be understood as media: after all, they are used to express the creative message of an author to a given audience.

The (methodological) depth of this case study lies firmly in the humanities. In analyzing PL design documents as media, CS methodologies are deeply insufficient. We wish to analyze and interpret a small, related set of documents in detail: a close reading. Drawing on this insight, the lesson turns to a standard technique from media studies: close reading. The broad technique of close reading comes in many specific forms, among which our analysis situates itself as a *hermeneutic inquiry* in the tradition of Schleiermacher [51, 54].

Hermeneutic inquiry arises originally from the interpretation of religious texts, and might be described as "reading as a form of divination." This method grapples with a fundamental challenge in textual interpretation: though an author expresses their own intentions in a text, the process of interpreting those meanings is fundamentally a creative and thus subjective process. Recognizing the creative aspect, the hermeneutic inquiry method does not aim for reproducible interpretations of a text. Instead, it engages with the author's creativity through the interpreter's own creativity. An emphasis is placed on an internal consistency of the interpretation, particularly a consistent interaction between the parts and the whole of the text, with language playing a pivotal role [54]. In the HCPL classroom, these insights are presented at a high level, emphasizing that we will pick apart the details of a text with focus on the meanings of specific details. We emphasize that we will employ subjectivity as we analyze the meanings and motivations behind these details.

We now undertake a close reading of the C+= design documents and prototype implementation in the tradition of hermeneutic inquiry, as is done in the course. We reveal how the documents promote both misogynistic and transphobic messages and how those messages are ultimately inseparable from one another. We quote several pieces of the text for analysis.

The prototype implementation consists of a series of C preprocessor macros performing simplistic keyword replacements; we list three example keywords which were used for classroom discussion.

```
#define privileged private
#define yell printf
#define social_construct class
```

Though this snippet is devoid of interesting technical content, its content as media is substantial. By placing these three definitions together, the snippet suggest these concepts are closely linked in the author's

mind both with each other and the concept of a feminist. This text suggests the author views feminists as likely to describe social categories as being constructed, e.g., drawing on Foucault or Butler, and that they are likely to discuss the topic of social privilege. More mundanely, the right-hand sides suggest an understanding of social categories as classes, private fields as privileged, and yelling as a form of I/O. The crucial line, however, is the definition of `yell`.

In placing these lines together, the author suggests that they understand discussions of privilege and social constructionism as a form of *yelling*. In the context of well-established tropes of online anti-feminist discourse, the (literal and figurative) keyword `yell` unmistakably evokes the trope of the hysterical feminist, who blames every smallest life inconvenience on patriarchy. This trope serves to erase all real, justified claims of gender inequality by flattening all speech by feminists into the category of hysteria.

Three (non-consecutive) snippets of the primary design document are also used for in-class analysis, quoted here [emphasis in original]:

> **Constants are not allowed**, as the idea of a lack of identity fluidity is problematic. ... any numeric value is a variable, and is required to take on at least 2 values over the course of the program, or the inHERpreter will throw a **Trigger Warning**.
>
> A number can be an integer or a double or a long if xir so identifies xirself.
>
> Booleans are banned for imposing a binary view of true and false. C+= operates paralogically and transcends the trappings of Patriarchal binary logic. **No means no, and yes could mean no as well.**

The transphobia of the first snippet requires some analysis, but is hard to argue against. The key term is *identity fluidity*, which is a frequently-used means of describing gender identity for some LGBT+ people, particularly genderfluid people. Juxtaposed against this, the following sentence is a clear metaphor for gender transition, with the requirement for taking on multiple values standing in for a requirement that a person take on multiple genders throughout life, i.e., a nonsensical requirement for all people to be transgender. The reference to trigger warnings further solidifies the trope of the hysterical feminist. By putting anti-trans and anti-feminist messages in the same paragraph, the author reveals their view of trans advocacy and feminist advocacy as fundamentally linked.

In the second snippet, the key words are *xir* and *xirself*. The words *xir* and *xirself* are *neopronouns*, i.e., neologisms chosen as personal pronouns, often for self-expression. Pronouns are ascribed to people, not programs, and by ascribing a neopronoun to a program, the text implies that its author believes that the desire of some queer people to play with language for self-expression is as ludicrous as calling a program a person. In my classroom, this concern is not an abstract one. Recall that Section 4.2 recounts a classroom interaction with a student who has neopronouns (specifically, ce/cer/cers). If my student encounted C+= in the wild, ce would rightfully interpret it as a personal attack. By making space for criticism of C+= discourse in the classroom, I concretely demonstrate to cer that ce will receive protection from cer academic community, the PL community, when attacked using our own jargon.

In the third snippet, against this backdrop, it is hard to see the discussion of Booleans as anything other than a reference to the gender binary. In its third sentence, *no means no* is a widely-understood slogan for advocating against sexual violence. Once again, the text links feminism and trans advocacy by juxtaposing the gender binary with references to sexual violence.

In summary, this proposal simultaneously contains many references to PL features that are laughable on their face to any experienced programmer, and consistently juxtaposes them will well-established concepts from feminism and especially transfeminism. In doing so, it seeks to build the association in its reader's mind that gender equality, and particularly trans equality, are equally laughable ideas.

This concludes the close reading performed in the lesson. By taking time to close-read this text as a class, we not only motivate the rest of the lesson, but demonstrate to students that humanistic techniques can give them, as computer scientists, a new and practical dimension of understanding that complements the methodologies with which they are more familiar.

5 Conclusion

This paper is a part of the *Human-Centered Programming Languages* (HCPL) project to develop an open-access course for the interdisciplinary study of programming languages. The project also includes an open-access textbook, course materials, and companion paper. The *humans* in HCPL represent different *scholars* of PLs, through five archetypes with different intellectual traditions: the Practitioner, the Implementer, the Theorist, the Social Scientist, and the Humanist. I call the HCPL course structure a *tour-of-humans* as opposed to a *tour-of-features* or *tour-of-paradigms*, because it is organized around these archetypes with an overarching goal of leaving students able to read and converse with each academic community. Though some students enter the classroom wondering how each academic tradition could fit into the topic of PL, just as many are glad for the opportunity to explore such connections for their first time. This paper was dedicated to exploring the role of case studies in HCPL. We gave detailed descriptions of a subset of the HCPL case studies: FLOW-MATIC, Processing, Twine, Torino, and C-Plus-Equality.

A key intellectual goal of this paper is to clearly separate the role of case studies in a *tour-of-paradigms* from their role in a *tour-of-humans*. The former usage has been widely criticized for many years as lacking intellectual depth. We thus arrive at the colloquial "thesis" of this paper, that PL case studies can be *deep*.

I made this thesis more precise: I illustrated how case studies serve as essential tools for teaching foundational concepts from the social sciences and humanities and also illustrated how these foundational concepts have clear implications for PL design. Educators of PL have long held interest in PL design, yet intellectually serious scholarship on design always needs techniques well-beyond the realm of CS, typically from both the social sciences and the humanities. When we view students as students of design, case studies are deep because they reveal insights that could not be found with *any other method*. As a student of design, engagement with actually-existing historical designs is central, not optional.

The HCPL curriculum is based in a fundamental assumption, which not all scholars might agree with: the foundations of PLs include the foundations of design in the social sciences and the humanities. This assumption has profound implications for pedagogy, of which the heavy use of case studies is but the surface. The assumption that design foundations *are* PL foundations radically alters our perspective of *who our courses are for*. I acknowledge this shift is a radical one, and that even sympathetic educators might sit with this an idea a long time before entertaining it in their own pedagogy.

Yet, I close by inviting the audience to sit with this idea for the precise reason that when we inspect who our courses are designed for, we are presented with a substantial opportunity for inclusive pedagogy. Many PL and FP educators believe in our hearts that our subject is fundamental, that it can be brought to application across an incredible array of topics, and that it is a topic for everyone. As curriculum designers, our work goes beyond making all people feel welcome in our classrooms. By engaging with a wide range of *scholarly traditions*, we may continue to broaden the range of students who find a deep and lasting connection with the topics we hold dear.

References

[1] Leif Andersen, Michael Ballantyne & Matthias Felleisen (2020): *Adding interactive visual syntax to textual code*. *Proceedings of the ACM on Programming Languages* 4(OOPSLA), pp. 1–28, doi:10.1145/3428290.

[2] Hugh H Benson (2006): *Plato's method of dialectic*. In: *Blackwell Companion to Plato*, Blackwell Publishing Malden, MA etc, Singapore, pp. 85–99, doi:10.1002/9780470996256.

[3] Rose Bohrer (2022): *Imagining Introductory Rust*. In: *RustEdu*, Rust Edu, Virtual, pp. 27–33.

[4] Rose Bohrer (2023): *Centering Humans in the Programming Languages Classroom: Building a Text for the Next Generation*. In Molly Q. Feldman & Michael Hilton, editors: *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E, SPLASH-E 2023, Cascais, Portugal, 25 October 2023*, ACM, pp. 26–37, doi:10.1145/3622780.3623646.

[5] Rose Bohrer (2023): *Human Centered Programming Languages*. Self-published. Available at `https://bookish.press/hcpl`.

[6] Virginia Braun & Victoria Clarke (2012): *Thematic analysis.* American Psychological Association, Washington, DC, doi:10.1037/13620-004.

[7] Mez Breeze (1994): *Mezangelle*. *Kajplats 305, and elsewhere.*

[8] Margaret M. Burnett, Simone Stumpf, Jamie Macbeth, Stephann Makri, Laura Beckwith, Irwin Kwan, Anicia Peters & Will Jernigan (2016): *GenderMag: A Method for Evaluating Software's Gender Inclusiveness*. *Interact. Comput.* 28(6), pp. 760–787, doi:10.1093/IWC/IWV046.

[9] micha cárdenas (circa 2013): *femme Disturbance library*. *Unpublished.* Because this work is unpublished, a precise date is unavailable.

[10] Victoria Clarke, Virginia Braun & Nikki Hayfield (2015): *Thematic analysis*. *Qualitative psychology: A practical guide to research methods* 3, pp. 222–248.

[11] Michael J. Coblenz, Ariel Davis, Megan Hofmann, Vivian Huang, Siyue Jin, Max Krieger, Kyle Liang, Brian Wei, Mengchen Sam Yong & Jonathan Aldrich (2020): *User-Centered Programming Language Design: A Course-Based Case Study*. *CoRR* abs/2011.07565. arXiv:2011.07565.

[12] Michael J. Coblenz, Gauri Kambhatla, Paulette Koronkevich, Jenna L. Wise, Celeste Barnaby, Joshua Sunshine, Jonathan Aldrich & Brad A. Myers (2021): *PLIERS: A Process that Integrates User-Centered Methods into Programming Language Design*. *ACM Trans. Comput. Hum. Interact.* 28(4), pp. 28:1–28:53, doi:10.1145/3452379.

[13] Lena Cohen, Heila Precel, Harold Triedman & Kathi Fisler (2021): *A New Model for Weaving Responsible Computing Into Courses Across the CS Curriculum*. In: *SIGCSE*, ACM, Virtual, pp. 858–864.

[14] Haris Delić & Senad Bećirović (2016): *Socratic method as an approach to teaching*. *European Researcher. Series A* 111(10), pp. 511–517.

[15] Carolyn Ellis, Tony E Adams & Arthur P Bochner (2011): *Autoethnography: an overview*. *Historical social research/Historische sozialforschung* 36(4), pp. 273–290.

[16] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt & Shriram Krishnamurthi (2018): *How to design programs: an introduction to programming and computing*. MIT Press.

[17] Shaun Ferns, Robert Hickey & Helen Williams (2021): *Ungrading, supporting our students through a pedagogy of care*. *International Journal for Cross-Disciplinary Subjects in Education* 12(2), pp. 4500–4504.

[18] Kathi Fisler, Shriram Krishnamurthi, Benjamin S. Lerner & Joe Gibbs Politz (2021): *A Data-Centric Introduction to Computing*. Self-published. Available at `dcic-world.org`.

[19] Feminist Software Foundation (2013): *C Plus Equality*. *GitHub*. Available at `https://github.com/TheFeministSoftwareFoundation/C-plus-Equality`.

[20] Interactive Fiction Technology Foundation (2023): *Twine 2.7.0*.

[21] Daniel P Friedman, Mitchell Wand & Christopher Thomas Haynes (2001): *Essentials of programming languages*. MIT press.

[22] Susan E George (2002): *Learning and the reflective journal in computer science*. In: *Australasian Computer Science Conference*, 2, Australian Computer Society, Melbourne, pp. 77–86.

[23] Rose M Giaconia & Larry V Hedges (1982): *Identifying features of effective open education*. *Review of educational research* 52(4), pp. 579–602, doi:10.3102/00346543052004579.

[24] Robert Harper (2016): *Practical foundations for programming languages*. Cambridge University Press, Cambridge, UK, doi:10.1017/CBO9781316576892.

[25] Felienne Hermans (2021): *The Programmer's Brain: What every programmer needs to know about cognition*. Manning, Shelter Island, NY.

[26] Grace Murray Hopper (1959): *Automatic programming: present status and future trends*. In: *Mechanisation of Thought Processes: Proceedings of a Symposium held at the National Physical Laboratory*, 1, pp. 155–194.

[27] Jan Hylén (2006): *Open educational resources: Opportunities and challenges*. *Proceedings of open education* 4963.

[28] Steve Klabnik & Carol Nichols (2023): *The Rust programming language*. No Starch Press, San Francisco.

[29] Amy J. Ko (2016): *What is a programming language, really?* In Craig Anslow, Thomas D. LaToza & Joshua Sunshine, editors: *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools, PLATEAU@SPLASH 2016, Amsterdam, Netherlands, November 1, 2016*, ACM, Amsterdam, pp. 32–33, doi:10.1145/3001878.3001880.

[30] Amy J. Ko & Brad A. Myers (2004): *Designing the Whyline: a debugging interface for asking questions about program behavior*. In Elizabeth Dykstra-Erickson & Manfred Tscheligi, editors: *Proceedings of the 2004 Conference on Human Factors in Computing Systems, CHI 2004, Vienna, Austria, April 24 - 29, 2004*, ACM, pp. 151–158, doi:10.1145/985692.985712.

[31] Alfie Kohn & Susan D Blum (2020): *Ungrading: Why rating students undermines learning (and what to do instead)*. West Virginia University Press, Morgantown.

[32] Dimitra Kokotsaki, Victoria Menzies & Andy Wiggins (2016): *Project-based learning: A review of the literature*. *Improving schools* 19(3), pp. 267–277, doi:10.1177/1365480216659733.

[33] Shriram Krishnamurthi (2003): *Programming languages - application and interpretation*. e-book. Available at `http://www.cs.brown.edu/%7Esk/Publications/Books/ProgLangs/`.

[34] Shriram Krishnamurthi (2008): *Teaching programming languages in a post-linnaean age*. *ACM SIGPLAN Notices* 43(11), pp. 81–83, doi:10.1145/1480828.1480846.

[35] Nicholas Laberge, K Hunter Wapman, Allison C Morgan, Sam Zhang, Daniel B Larremore & Aaron Clauset (2022): *Subfield prestige and gender inequality among US computing faculty*. *Communications of the ACM* 65(12), pp. 46–55, doi:10.1145/3535510.

[36] Imre Lakatos (1963): *Proofs and refutations*. Nelson, London.

[37] username Letum et al. (2014): *C Plus Equality C+=*. *Know Your Meme*. Available at `https://knowyourmeme.com/memes/events/c-plus-equality-c`. The author is aware that some readers might balk at the deeply non-academic nature of this source. Because I seek to analyze online interactions which occurred in a deeply non-academic community, critical analysis of non-academic sources is often the only viable option. Accessed: December 15, 2023.

[38] T. Memmott (2011): *Codework: Phenomenology of an anti-genre*. *Journal of Writing in Creative Practice* 4(1), pp. 93–105, doi:10.1386/jwcp.4.1.93_1.

[39] John C Mitchell (1996): *Foundations for programming languages*. 1, MIT Press, Cambridge, MA.

[40] Jenny Morales, Cristian Rusu, Federico Botella & Daniela Quiñones (2019): *Programmer eXperience: A Systematic Literature Review*. *IEEE Access* 7, pp. 71079–71094, doi:10.1109/ACCESS.2019.2920124.

[41] Cecily Morrison, Nicolas Villar, Anja Thieme, Zahra Ashktorab, Eloise Taysom, Oscar Salandin, Daniel Cletheroe, Greg Saul, Alan F. Blackwell, Darren Edge, Martin Grayson & Haiyan Zhang (2020): *Torino: A Tangible Programming Language Inclusive of Children with Visual Disabilities*. *Hum. Comput. Interact.* 35(3), pp. 191–239, doi:10.1080/07370024.2018.1512413.

[42] Brad A. Myers, Amy J. Ko, Thomas D. LaToza & YoungSeok Yoon (2016): *Programmers Are Users Too: Human-Centered Methods for Improving Programming Tools*. *Computer* 49(7), pp. 44–52, doi:10.1109/MC.2016.200.

[43] Brad A. Myers, John F. Pane & Amy J. Ko (2004): *Natural programming languages and environments*. *Commun. ACM* 47(9), pp. 47–52, doi:10.1145/1015864.1015888.

[44] Graham Nelson (2001): *The Inform Designer's Manual*. Interactive Fiction Library, Virtual.

[45] Lene Nielsen (2013): *Personas - User Focused Design*. *Human-Computer Interaction Series* 15, Springer, London, doi:10.1007/978-1-4471-4084-9.

[46] Robert Nystrom (2021): *Crafting interpreters*. Genever Benning.

[47] Benjamin C Pierce (2002): *Types and programming languages*. MIT Press, Cambridge, MA.

[48] Benjamin C Pierce (2004): *Advanced topics in types and programming languages*. MIT Press, Cambridge, MA, doi:10.7551/mitpress/1104.001.0001.

[49] Casey Reas & Ben Fry (2006): *Processing: programming for the media arts*. *Ai & Society* 20, pp. 526–538, doi:10.1007/s00146-006-0050-9.

[50] John C Reynolds (1998): *Theories of programming languages*. Cambridge University Press, Cambridge, UK, doi:10.1017/CBO9780511626364.

[51] Friedrich Schleiermacher (1998): *Schleiermacher: hermeneutics and criticism: and other writings*. Cambridge University Press, doi:10.1017/CBO9780511814945.

[52] Ari Schlesinger (2014): *A Feminist Programming Language*. *FemTechNet*. Follow-up article to original blog post.

[53] John Seely Brown & RP Adler (2008): *Open education, the long tail, and learning 2.0*. *Educause review* 43(1), pp. 16–20.

[54] David G Smith (1991): *Hermeneutic inquiry: The hermeneutic imagination and the pedagogic text*. *Forms of curriculum inquiry* 3.

[55] Katta Spiel, Kathrin Gerling, Cynthia L Bennett, Emeline Brulé, Rua M Williams, Jennifer Rode & Jennifer Mankoff (2020): *Nothing about us without us: Investigating the role of critical disability studies in HCI*. In: *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pp. 1–8, doi:10.1145/3334480.3375150.

[56] International Organization for Standardization (2018): *Ergonomics of human-system interaction, Part 11: Usability: Definitions and concepts*. Standard, International Organization for Standardization, Geneva, CH.

[57] Jesse Stommel (2018): *How to ungrade*. Available at `https://www.jessestommel.com/how-to-ungrade/`.

[58] Alaaeddin Swidan & Felienne Hermans (2023): *A Framework for the Localization of Programming Languages*. In Molly Q. Feldman & Michael Hilton, editors: *Proceedings of the 2023 ACM SIGPLAN International Symposium on SPLASH-E, SPLASH-E 2023, Cascais, Portugal, 25 October 2023*, ACM, pp. 13–25, doi:10.1145/3622780.3623645.

[59] Remington Rand Univac (1957): *FLOW-MATIC PROGRAMMING SYSTEM*. Technical Report, Sperry Rand Corporation.

[60] Molly White (2013): *Why I'm not laughing at C Plus Equality*. *Self-publised*. Available at `https://blog.mollywhite.net/why-im-not-laughing-at-c-plus-equality/`. Accessed: December 15, 2023.

[61] David Wiley, TJ Bliss & Mary McEwen (2014): *Open educational resources: A review of the literature*. In: *Handbook of research on educational communications and technology*, Springer, New York, NY, pp. 781–789, doi:10.1007/978-1-4614-3185-5_63.

[62] Katherine Ye, Wode Ni, Max Krieger, Dor Ma'ayan, Jenna Wise, Jonathan Aldrich, Joshua Sunshine & Keenan Crane (2020): *Penrose: from mathematical notation to beautiful diagrams*. *ACM Trans. Graph.* 39(4), p. 144, doi:10.1145/3386569.3392375.

[63] Barry J Zimmerman (2002): *Becoming a self-regulated learner: An overview*. Theory into practice 41(2), pp. 64–70, doi:10.1207/s15430421tip4102_2.